CAMPUS
DE EXCELENCIA
INTERNACIONAL

Graduado en Ingeniería Informática
Universidad Politécnica de Madrid

Facultad de Informática

TRABAJO FIN DE GRADO

OPTIMIZACIÓN Y PARALELIZACIÓN DE UN ALGORITMO DE SINCRONIZACIÓN MEDIANTE EL USO DE GPUS Y LA TECNOLOGÍA CUDA

Autor: Ignacio Molina Cuquerella
Director: Francisco Javier Rosales García

MADRID, ENERO DE 2013

Gracias a Dopi, Fran, Maribel y Marisa por la oportunidad de trabajar en el grupo GOPAC con ellos y gracias a toda la gente del departamento que siempre me ha hecho sentir como uno mas.

Muchas gracias a Eli por soportarme durante todos estos años a pesar de que sé que a veces no soy el mejor compañero de practicas.

Finalmente quería dar las gracias a mis padres y a mi hermano por su apoyo incondicional.

Dedico este trabajo a mi hermana,
Asunción Molina Cuquerella.
En mi mente siempre

ÍNDICE

RESUMEN / ABSTRACT

En el presente documento se hablara acerca del desarrollo de un proyecto para la mejorara de un programa de análisis de señales; con ese fin, se hará uso de técnicas de optimización del software y de tecnologías de aceleración, mediante el aprovechamiento del paralelismo del programa.

Además se hará un análisis de acerca del uso de dos tecnologías basadas en diferentes paradigmas de programación paralela; una mediante múltiples hilos con memoria compartida y la otra mediante el uso de GPUs como dispositivos de co-procesamiento.

This paper will talk about the development of a Project to improve a program that does signals analysis; to that end, it will make use of software optimization techniques and acceleration technologies by exploiting parallelism in the program.

In Addition will be done an analysis on the use of two technologies based on two different paradigms; one using multiple threads with shared memory and the other using GPU as co-processing devices.

1. INTRODUCCIÓN

1.1. ¿PARA QUE SIRVE OPTIMIZAR UN PROGRAMA?

Optimizar 1. tr. Buscar la mejor manera de realizar una actividad.
[Real Academia Española]

A la hora de crear un programa el objetivo principal es que este realice la tarea deseada correctamente, es por ello que la mayoría de programadores cuando alcanzan este objetivo dan el trabajo por concluido; pero cuando un programa esta destinado a un uso intensivo, el tiempo en realizar la tarea se vuelve un factor clave. En ese punto optimizar un programa es fundamental.

En la informática hay muchas maneras de logran realizar una misma tarea; a la hora de optimizar un programa, encontrar la mejor forma es una tarea ardua y compleja; en la que se debe aplicar un profundo conocimiento tanto de la arquitectura hardware y como del software, pero puede suponer una mejora significativa con respecto al programa original.

1.2. ¿QUÉ SIGNIFICA LA PROGRAMACIÓN PARALELA?

Existen varios métodos de lograr acelerar un programa y uno de ellos es la programación paralela; consistente en dividir una tarea en varias tareas mas pequeñas e independientes entre si, y ejecutarlas de forma concurrente. En un escenario ideal se podría conseguir que el tiempo dedicado a una tarea se redujera proporcionalmente en relación con el numero de trabajadores concurrentes dedicados a ella. Aunque en escenarios reales muchas veces se logra ese ideal, si que se consigue en el programa paralelo un beneficio notable con respecto al mismo ejecutando de forma secuencial.

1.3. OBJETIVO DEL PROYECTO

El objetivo de este trabajo de fin de grado es trasladar un programa Matlab, de análisis de datos, al lenguaje C; el cual es un lenguaje de programación mas rápido y de bajo nivel, lo que perimirá una mayor optimización del programa respecto al hardware.

Otro de los objetivos del proyecto es tratar de paralelizar la tarea que realiza el programa mediante el uso de dos tecnologías con visiones diferentes, y actualmente en auge en el ámbito del computo científico. OpenMp, el cual se basa en un sistema multiproceso de memoria compartida, y CUDA, cuya mecánica radica en el uso de GPUs como unidas de co-procesamiento matemático.

1.4. ESTRUCTURA DEL DOCUMENTO

Es documento se estructura de la siguiente forma.

Capitulo 2: Se hará una explicación de para qué sirve y como funciona el algoritmo Synchronization Likelihood.
Capitulo 3: En este capitulo presenta el estado actual de las tecnologías de paralelización y se hace una breve explicación de su funcionamiento.
Capitulo 4: Se contara mediante un desglose por etapas como ha sido el desarrollo del proyecto.
Capitulo 5: Se establecerán unas conclusiones globales en los cuales se expondrán los beneficios y dificultades de las tecnologías usadas.

2. SYNCHRONIZATION LIKELIHOOD

2.1. ACERCA DEL ALGORITMO

El algoritmo Synchronization Likelihood[1] al cual me referiré a partir de este punto como algoritmo SL, busca hallar la correlación que existe entre un conjunto de canales de datos de muestras discretas, registrados de las ondas cerebrales de los pacientes mediante el uso de un dispositivo con forma de casco, compuesto por varios cientos de sensores magnéticos, durante un periodo continuado de tiempo; de forma que pueda verse el efecto de un evento ocurrido en un canal (sensor) sobre el resto de canales dentro de una ventana acotada de tiempo. Esto permite que se puedan provocar estímulos sobre un paciente y ver que áreas del cerebro se ven afectadas.

Mediante el uso del algoritmo SL y otros similares, los investigadores médicos disponen de las herramientas para poder comprender mejor el funcionamiento del cerebro humano y como interactúan las distintas regiones del cerebro, y con un mayor entendimiento de en un futuro podrían llegar a detectarse enfermedades neurodegenerativas como el Alzheimer en etapas tempranas de su desarrollo, posibilitando el tratamiento o incluso su curación.

Las principales dificultades a la que se enfrentan los investigadores a la hora de usar este tipo de algoritmos, es que éstos de por si son muy lentos, pero además tienen que manejar gran cantidad de datos para tan solo unos segundos de muestras. Lo cual dificulta mucho la gran mayoría de experimentos y fuerza a los investigadores a usar solo aquellos algoritmos mas ligeros y rápidos, pero también menos fiables y precisos.

2.2. DATOS DE ENTRADA

Los datos de entrada para el algoritmo es un fichero de texto plano en el cual se muestran los datos recogidos por un magneto-encefalograma durante un periodo continuado de tiempo.

Los datos están representados de forma que cada fila muestra la captura en un instante de tiempo de las señales recibidas por los distintos canales, y los canales están representados por columnas.

2.3. MÉTODO

Tras estudiar el algoritmo he concluido que este puede dividirse en dos fases bien diferenciadas.

2.3.1. ANÁLISIS DE UN CANAL

Para cada instante de tiempo *i* de los datos de entrada *I* de cada canal *K* debemos hacer lo siguiente.

Para cada vector formado por la señal actual y cada *L* elementos del canal hasta un máximo de *M*; se calcula la distancia euclidea dentro del mismo canal, entre éste y todos los vectores igualmente formados, dentro de un intervalo de tiempo.

Este intervalo se define con dos valores: *W*, que representa la distancia máxima en el pasado y en el futuro entre las que se realiza este cálculo de distancias; y *X*, que nos define un margen de exclusión para evitar el riesgo de auto-correlación (*Theiler correction for autocorrelation*), hacia delante y hacia atrás, entre el instante al actual, al que llamaremos presente; y el primer tiempo sobre el que calcular la distancia.

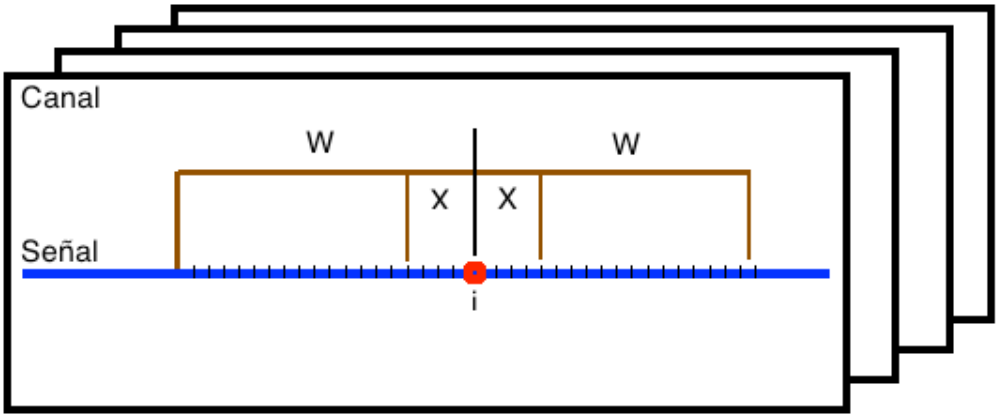


*Fig. 1 - Intervalos alrededor de un instante, definidos por **W** y **X**.*

Una vez calculadas todas las distancias dentro de las ventanas definidas por *W* y *X*, nos quedamos solo con aquellas *N* más próximas, siendo el valor N obtenido como el número de muestras dentro de la ventana por un valor arbitrario de probabilidad *P*, parámetro del algoritmo que no debería ser nunca mayor a 0.5.

$$N = P \times 2 \times (W - X)$$

Una vez hallados los *N* tiempos mas próximos en cada canal, para cada ventana de tiempos; pasamos a la siguiente fase.

2.3.2. COMPARACIÓN DE CANALES

Para cada instante de tiempo *i* de los datos de entrada *I* de cada canal *k* debemos hacer lo siguiente.

En esta fase comparamos cada canal con los demás, buscando que entre dos canales, para un mismo tiempo, se hayan encontrado a la misma distancia, dentro de la ventana *W*, muestras dentro del rango de las *N* próximas; siendo así, contabilizamos una coincidencia entre ambos canales.

Una vez halladas todas las coincidencias para todos los canales en todo el periodo de tiempo a analizar, se pondera por el numero de tiempos analizados.

2.3.3. OBSERVACIONES

Como la ventana W examina datos hacia delante y hacia atrás en el tiempo con respecto al instante de tiempo que se esta analizando en cada momento, surgen dos secciones de conflicto que hay que tener en cuenta. Primeramente, al empezar el algoritmo, cuando aun no hay valores en "pasado"; y en segundo lugar al final, cuando no hay mas "futuro". Para mi proyecto, la solución establecida es ir reduciendo un lado de la ventana cuando nos acercamos a alguno de estos límites.

2.4 ANTERIORES ADAPTACIÓNES DEL ALGORITMO

Antes de empezar a trabajar en este proyecto existía una implementación previa del algoritmo SL realizada en Matlab, ésta versión era muy lenta por lo que su uso no era practico para los investigadores. Esta implementación se tomo en un primer momento como punto de partida para la adaptación al lenguaje de programación C, sobre la que voy a hablar en el presente documento.

Por ello, en las comparativas de tiempos que haga según evoluciona mi programa hare referencia a esta versión Matlab como punto de origen.

3. TECNOLOGÍAS DE PARALELIZACION EN LA ACTUALIDAD

3.1. PARALELIZACIÓN TRADICIONAL

Tradicionalmente en el ámbito científico cuando se busca la paralelizar un programa se recurre a mecanismos como Message Passing Interface, también conocido como MPI, el cual consiste en el intercambio de información entre procesos distribuidos en una red de nodos interconectados; o al uso de múltiples procesos o hilos dentro de un mismo nodo, aprovechándose de la capacidad de ejecución paralela de algunos procesadores.

En el primer caso, MPI requiere de una infraestructura con múltiples nodos y un una rápida conexión de datos entre ellos, lo cual suele ser bastante caro; mientras que en el segundo, tan solo se requiere de una maquina con múltiples núcleos de procesamiento, los cual a día de hoy es bastante habitual en cualquier equipo de gama media.

3.1.1. ¿QUÉ ES OPENMP?

OpenMP[10] es una interfaz de programación de aplicaciones multiproceso de memoria compartida, actualmente, estándar en la mayoría de compiladores. Esta basado en una serie de directivas que, en tiempo de compilación, son usadas por el compilador para introducir el código necesario para lanzar al mismo tiempo múltiples hilos; los cuales, se dividen el trabajo de la región paralela según lo haya definido el programador y luego unifican los resultados.

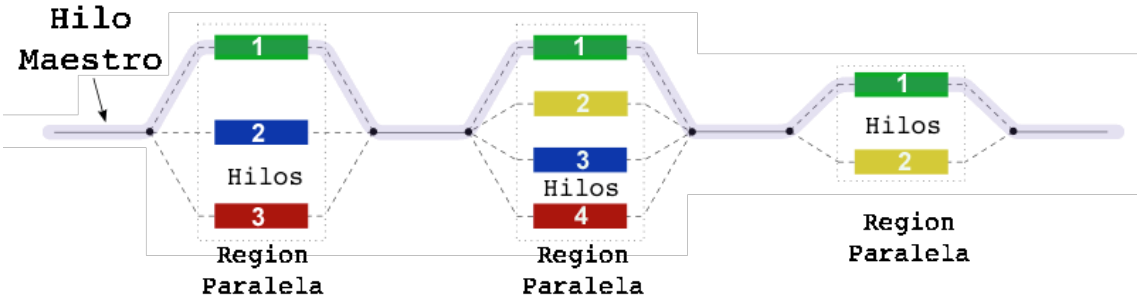


Fig. 2 - Lanzamiento de múltiples hilos en regiones paralelas.

OpenMP es el resultado de la unión de varios proveedores de hardware y software para lograr un modelo de programación portable y fácilmente escalable, que al mismo tiempo pueda resultar simple y flexible para los desarrolladores de aplicaciones paralelas.

Esta interfaz permite mediante directivas un fácil manejo de los hilos, distintos tipos de esquemas de planificación trabajo y un lenguaje claro para la declaración de

variables compartidas o privadas de cada hilo; así como también dispone de múltiples operadores para la agrupación de resultados.

Además es compatible con otros modelos de programación paralela, como la ya mencionada MPI.

Aunque OpenMP es un estándar, sigue evolucionando constantemente con cada nueva versión, añadiendo nuevas funcionalidades requeridas por los desarrolladores y que rápidamente son incorporadas en los compiladores mas utilizados.

3.1.2. EJEMPLO DE ESCALABILIDAD

A continuación voy a mostrar un ejemplo de un código, en el lenguaje de programación C, de multiplicación de matrices con y sin OpenMP; para demostrar la simplicidad de su uso.

Codigo sin OpenMP

```
int i;
for (i=0;i<numFilas1;i++)
{
   int j;
   for (j=0;j<numFilas2;j++)
   {
        double acumulador = 0;
        int k;
        for (k=0;k<numColumnas1;k++) {
           acumulador += m1[i+k*numColumnas1] * m2[j*numColumnas2+k];
        }
        resultado[i*numFilas1+j] = acumulador;
   }
}
```

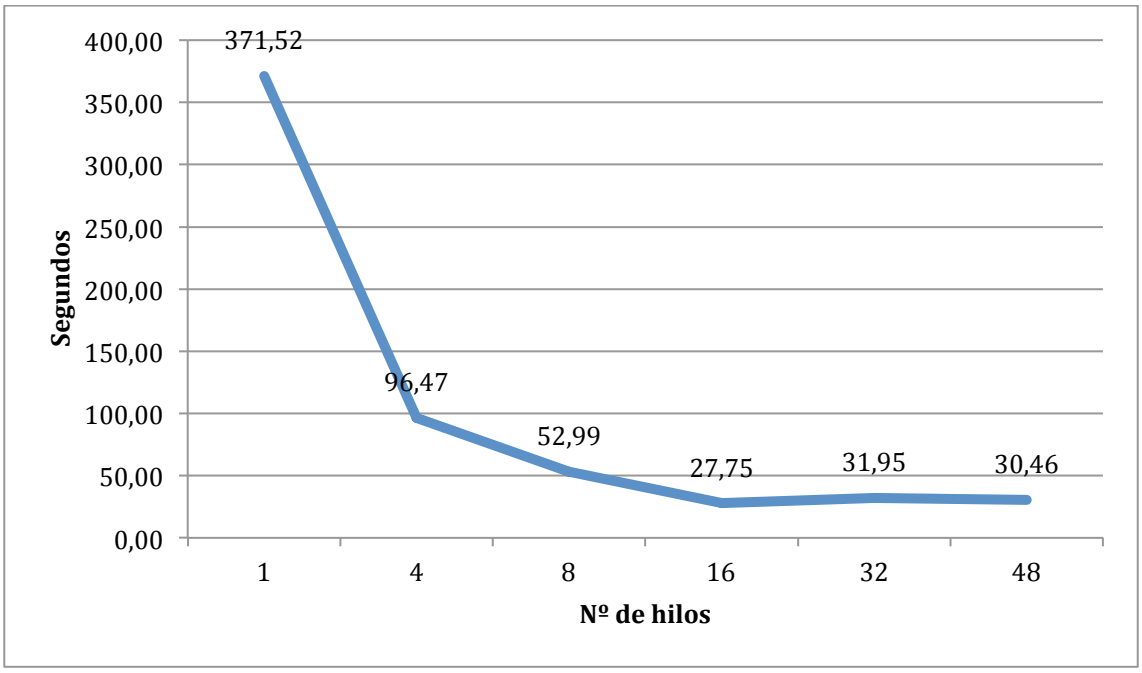
Codigo con OpenMP

```
int i;
#pragma omp parallel for default(shared)
for (i=0;i<numFilas1;i++)
{
   int j;
   #pragma omp parallel for
   for (j=0;j<numFilas2;j++)
   {
        double acumulador = 0;
        int k;
        for (k=0;k<numColumnas1;k++) {
           acumulador += m1[i+k*numColumnas1] * m2[j*numColumnas2+k];
```

```
        }
        resultado[i*numFilas1+j] = acumulador;
    }
}
```

Como puede verse no hay casi diferencias en el código, a excepción de unas pocas directivas de compilación con la clausula '*omp*'.

En la siguiente grafica podemos comparar el rendimiento paralelo del ejemplo, usando OpenMP con distinto numero de hilos, utilizando como entorno para el experimento la maquina Magnolio referida en el anexo del presente documento.



Graf. 1 - Comparativa de tiempos multiplicación de matrices de 5000x1000.

Como puede observarse en la grafica a medida que vamos usando mas hilos, el programa tarda menos en realizar la misma tarea; desde 2 a 16 hilos, el aumento en la velocidad se mantiene proporcional al numero de hilos involucrados en la tarea, pero a partir de 16 aumentar el numero de hilos no logra acelerar la tarea o incluso se ve ralentizada. Este comportamiento puede deberse a múltiples factores, como por ejemplo, la aparición de fallos de cache o una mala planificación del trabajo de cada hilo.

Con esto quiero dar a entender que aunque la interfaz OpenMP dota al programador de un mecanismo simple y flexible a la hora de trabajar con programas multi-hilo, no evita que sea necesario un estudio profundo del problema a tratar para poder resolver los diversos problemas que surgen al realizar una tareas con múltiples hilos.

3.2. PARALELIZACION EN GPU

3.2.1. ¿QUÉ ES UNA GPU?

Una GPU o **G**raphic **P**rocessing **U**nit es un dispositivo co-procesamiento grafico muy habitual en cualquier ordenador personal, que durante años ha estado asociado al sector de los videojuegos. En los últimos años, con el avance en la arquitectura de computadores y el desarrollo de GPUs cada vez mas rápidas y con mayor capacidad de computo, un nuevo nombre ha surgido para referirse a las GPUs de ultima generación, **G**eneral-**P**urpose computing **G**raphic **P**rocessing **U**nits también llamadas GPGPU; en respuesta a un nuevo paradigma en el uso de estos dispositivos.

Los desarrolladores comprendieron que, el gran numero de unidades aritmético-lógicas altamente especializadas que hasta ahora solo se habían considerado para el procesado de gran cantidad de vértices para la representación de figuras en tres dimensiones; eran, de facto, plataformas masivamente paralelas que podían ser usados también para procesar gran cantidad de otro tipo de datos. Es por ello que, tras ver los múltiples intentos de programadores independientes para hacer uso de esa capacidad de computo escondida en las GPUs y augurando una posibilidad de mercado, empresas como Nvidia o ATI (ahora AMD) empezaron a desarrollar herramientas para abrir el acceso a sus dispositivos para aquellos programadores que estuvieran interesados.

Para el desarrollo de este proyecto voy ha hacer uso de la herramienta CUDA desarrollada por Nvidia, mi elección se ha debido a varios factores siendo el primero y mas importante el hecho de que la propuesta de Nvidia es la mas extendida, con una comunidad mas grande; y por tanto me permite una mayor facilidad para encontrar la solución a los diversos problemas que puedan surgir; otro factor, fue la gran cantidad de documentación escrita por la propia Nvidia acerca del funcionamiento de la arquitectura de los dispositivos de Nvidia y CUDA[3].

3.2.3. ¿QUÉ ES CUDA?

Al igual que muchas otras empresas dedicadas al desarrollo de GPUs, Nvidia desarrolló un conjunto de herramientas que posibilitan a cualquier programador exprimir el potencial de sus GPUs y las dejó gratuitamente a disposición de los usuarios, a este conjunto de herramientas se denominó con el nombre de CUDA, por las siglas en ingles de **C**ompute **U**nified **D**evice **A**rchitecture.

CUDA esta compuesto por un lenguaje de programación similar a C, un compilador propio, y varias herramientas de depuración; además es compatible con múltiples lenguajes de programación de uso generalizado como C, C++, Python, Fortran o Java.

Mediante CUDA, un programador, puede crear programas capaces de comunicarse con las GPGPUs de Nvidia para mandarle tareas, los cuales reciben el nombre de Kernel.

3.2.3. ARQUITECTURA DE UNA GPU NVIDIA

Programar para GPUs requiere de una curva de aprendizaje medianamente larga, esto es debido a que la arquitectura de una GPU funciona diferentemente a como lo hace una CPU, lo cual requiere que el programador se replantee la forma en la que haría normalmente una tarea.

3.2.3.1. TRANSFERENCIA DE DATOS

Antes de empezar a explicar como funciona internamente una GPU, quiero dejar claro como se comunican estos dispositivos con la CPU y con la memoria principal de un ordenador, pues esto es uno de los factores mas importantes a tener en cuenta a la hora de trabajar con GPUs.

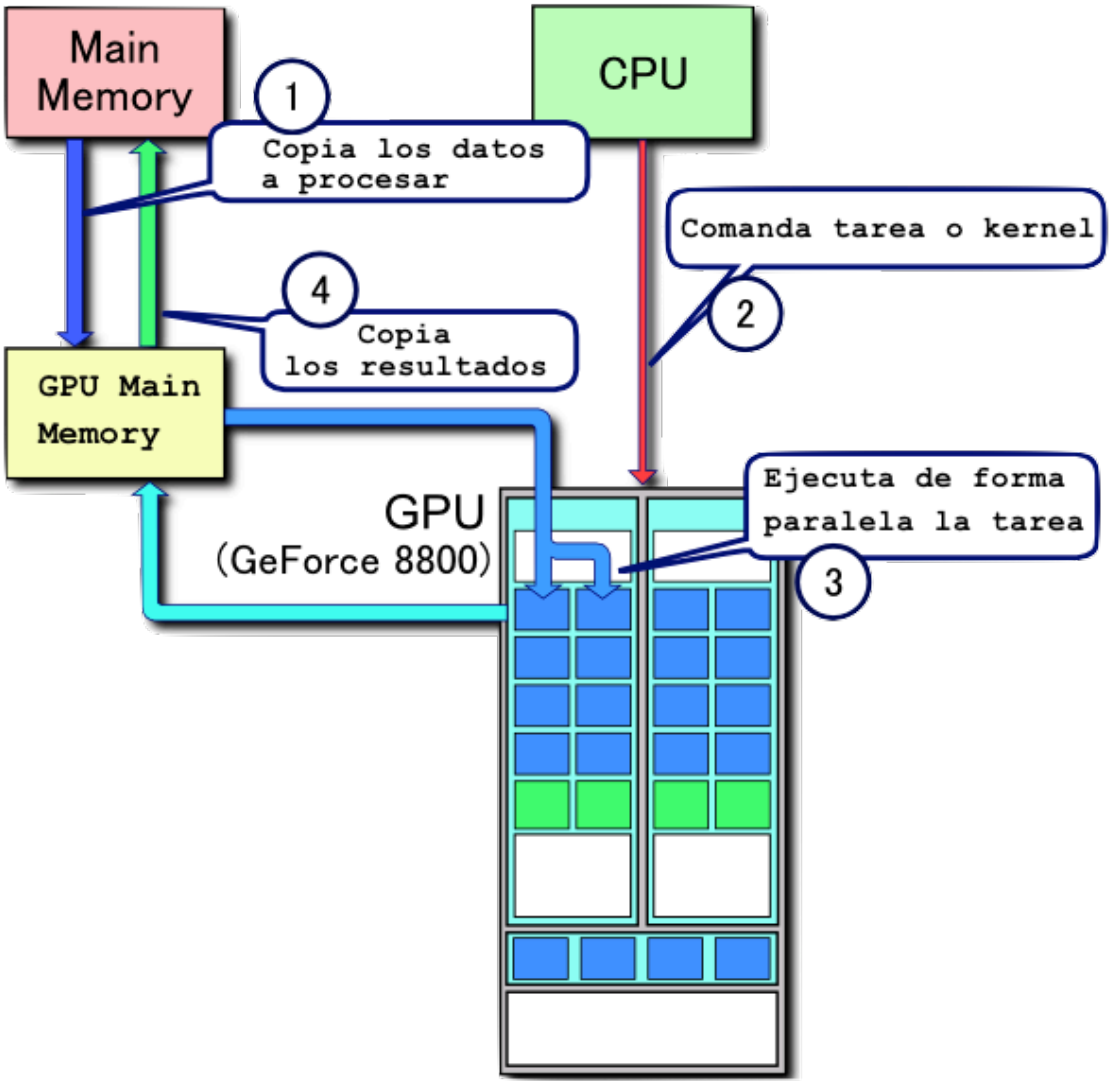


Fig. 3 - Flujo de trabajo de una ejecución con CUDA.

Como puede apreciarse en la anterior imagen, para poderse ejecutar un trabajo en una GPU primeramente debe transferirse, desde la memoria principal del ordenador a la memoria principal del dispositivo, los datos que se van a procesar; el Kernel solo trabajara en la GPU sin acceder a la memoria principal del ordenador, por lo que de igual forma los resultados deberán copiarse una vez finalizada la ejecución a la memoria principal. Para ello, CUDA dispone de métodos que permiten al programador reservar memoria y copiar datos desde la CPU, al cual a partir de ahora nos referiremos como Host.

Debido al elevado coste de tiempo que supone estas transferencias de datos entre el Host y la GPU es recomendable, sino imperativo, que en el caso de usar varios Kernel seguidos se reutilice la información ya copiada en la GPU y se evite, al máximo, el numero de transferencias.

Como ya he mencionado es importante mantener los datos del problema en la GPU de forma persistente durante la ejecución, el tamaño de la memoria principal varia según los modelos siendo aquellos con mas capacidad los modelos mas caros, pero incluso en aquellos con mayor memoria, su capacidad sigue siendo limitada, por lo que es primordial para un programador ser consciente de cuanta memoria esta ocupando y tratar de optimizarla al máximo.

Una vez explicado como se asigna una tarea y la importancia de mantener los datos presentes en la GPU y tener en cuenta el uso de memoria, pasare a explicar el funcionamiento y la distribución de hilos en una GPU.

3.2.3.2 Organización de los hilos a nivel logico

A nivel lógico los hilos de un Kernel se dividen en dos niveles, cada uno de los cuales esta ordenado en tres dimensiones para facilitar la distribución de tareas según el identificador del hilo.

En el nivel mas bajo los hilos están organizados en bloques, pudiendo repartirse mediante coordenadas en tres dimensiones *(x, y, z)*. A su vez cada bloque esta repartido en las coordenadas *(x, y, z)* de la malla del Kernel. Esta distribución es muy flexible y facilita la identificación de cada hilo y la asignación de tareas que debe realizar.

3.2.3.3 ORGANIZACIÓN DE LOS HILOS A NIVEL FISICO

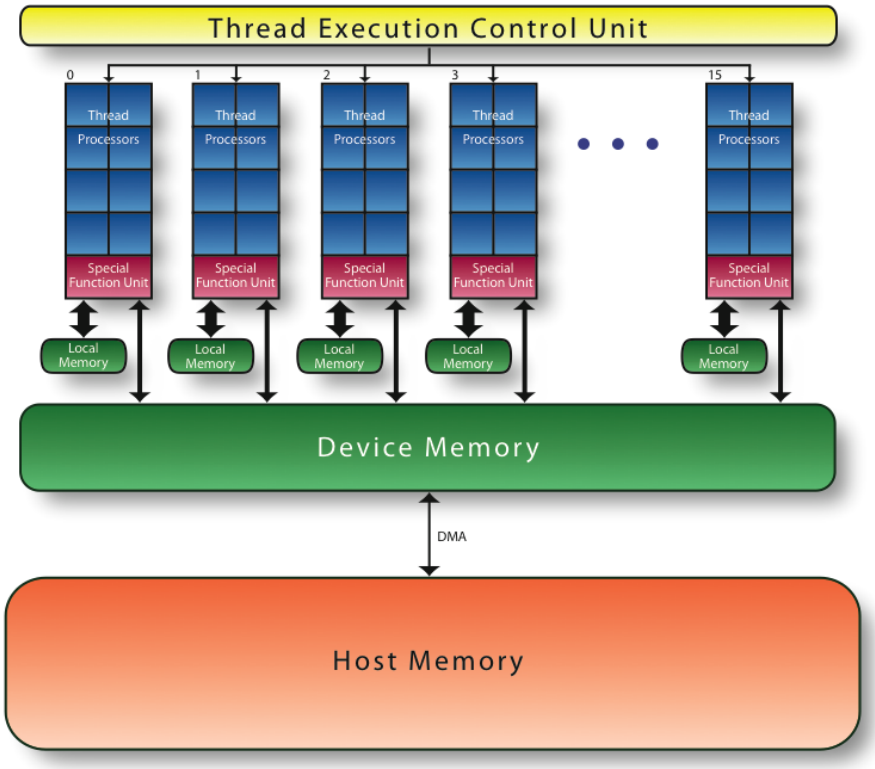


Fig. 4 - Componentes GPU Nvidia.

Los dispositivos de Nvidia cuentan con una memoria principal, una unidad de control de hilos y una serie de multiprocesadores, a los cuales nos referiremos a partir de ahora como SM por su nombre en ingles **S**treaming **M**ultiprocessors; además hay una cache L2 entre la memoria principal de la GPU y los SM, a la que acceden todos ellos. La unidad de control se encarga de la distribución de los hilos entre los distintos SM, teniendo en cuenta su identificador lógico, de forma que hilos de un mismo bloque permanecen siempre en el mismo SM para mantener la coherencia de caches.

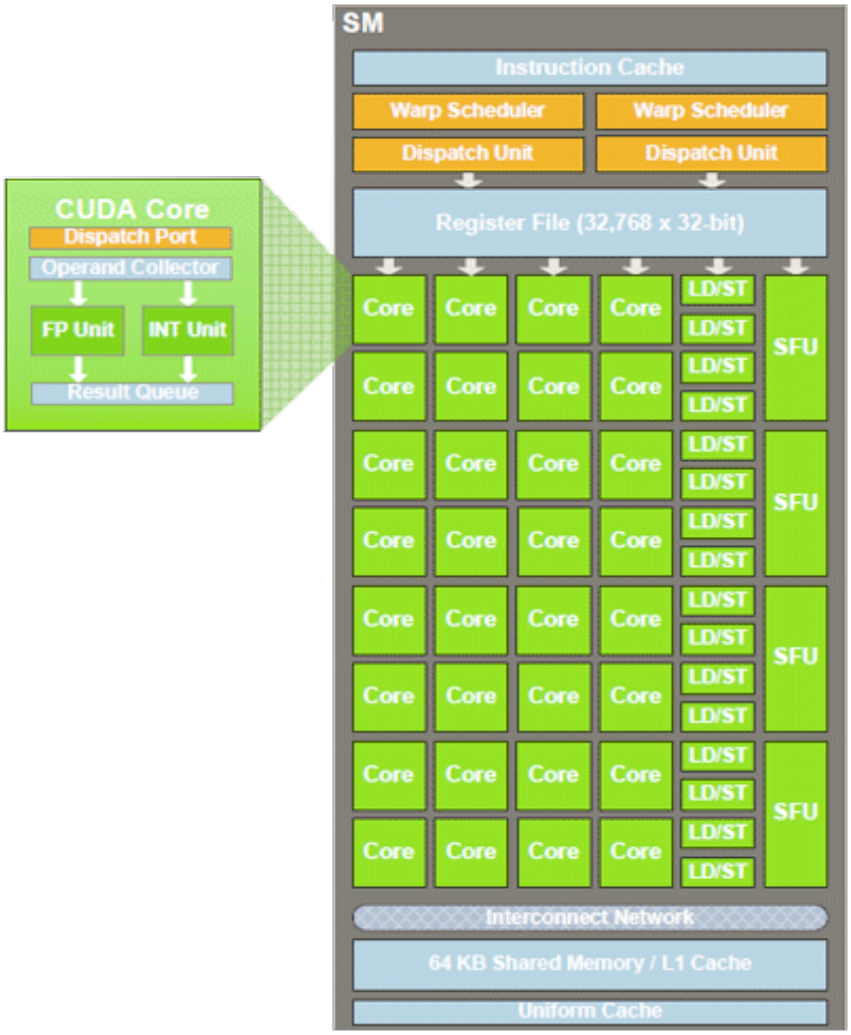


Fig. 5 - Estructura de un SM en la arquitectura Fermi[2].

La estructura de un SM en la arquitectura Fermi esta compuesto principalmente por 32 unidades de proceso llamados CUDA Core y que en conjunto son llamados *wrap*, una cache de instrucciones, una cache de datos L1, 16 dispositivos de acceso a memoria, una unidad de registros a la que tiene acceso cada uno de los Core del SM, y cuatro unidades para operaciones especiales.

Los SM están basados en SIMT o **S**ingle **I**nstruction **M**ultiple **T**hreads, modelo similar al vectorial SIMD usado en CPUs, según el cual cada uno de los 32 CUDA Core de un SM ejecuta exactamente la misma instrucción; pero usando los identificadores de hilo cada uno puede afectar a distintos datos de memoria o registros. Un programador de CUDA debe tener esta idea en mente en todo momento, porque si creamos un Kernel con divergencias, esto es, usando estructuras tipo *If-else*, obliga a que de los 32 Core primero ejecuten aquellos que fluyen por una rama y luego los de la otra, de forma que muchos Core permanecen ociosos y se desperdicia gran capacidad de computo; es por ello que a la hora de plantear transportar un programa a CUDA es importante, como primera etapa, tratar de eliminar todos los puntos de divergencia o trasladarlas a una

capa superior del programa; si esto no es posible, probablemente CUDA no sea la mejor solución para el problema.

Un CUDA Core es un procesador muy simple y especializado que solo dispone de una unidad aritmético lógica para enteros y otra para números de coma flotante.

He mencionado que hay 32 CUDA Core y 16 dispositivos de acceso a memoria, esto es debido a que los accesos se realizan en medio ciclo, permitiendo que tras un ciclo entero todos los cores hayan podido realizar el acceso. Como cada hilo del SM comparte una memoria cache y realizan el mismo trabajo, es importante que la alineación de los datos en memoria sea tal que los 32 CUDA Core del *wrap* accedan a datos contiguos de un mismo bloque, para asegurar un SM puede acceder a todos los datos con una sola transacción.

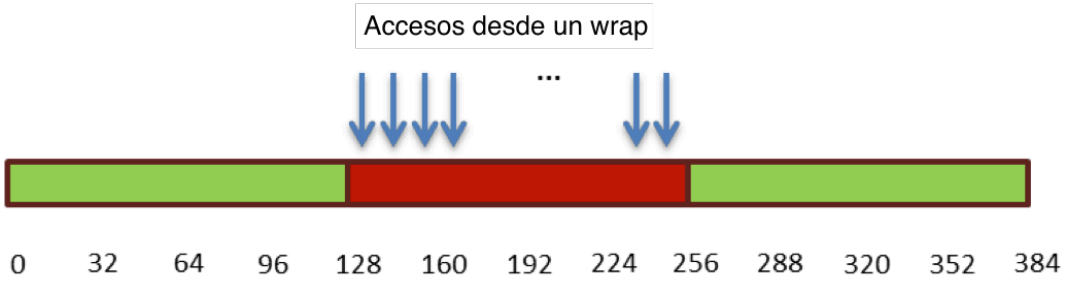


Fig. 6 - Acceso alineado a bloque.

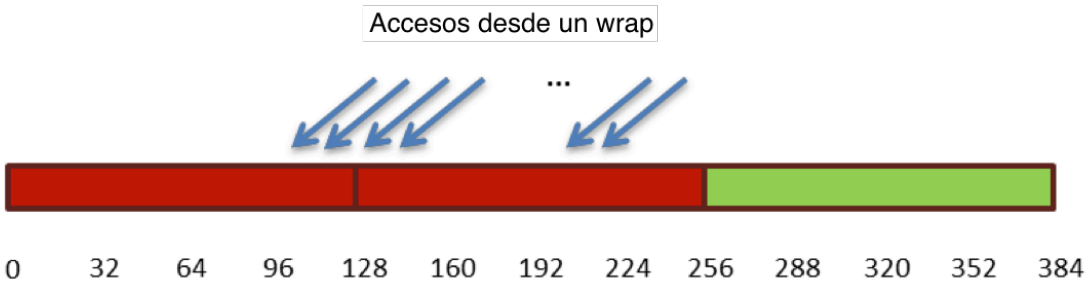


Fig. 7 - Acceso no alineado a bloque.

La cache L1 es mucho mas reducida en capacidad y mas rápida que la cache L2, esta compartida por todos los cores del SM y además cuenta con un pequeño rango al que puede acceder el programador de forma manual, a costa de perder capacidad en la L1, llamada memoria compartida.

Los registros son el tipo de memoria mas rápida de la que constan las GPU, aunque su uso esta compartido por cada uno de los Core del SM, su acceso es privado de cada hilo; por ello, cuantos mas registros use un Kernel menor numero de hilos podrán ejecutarse.

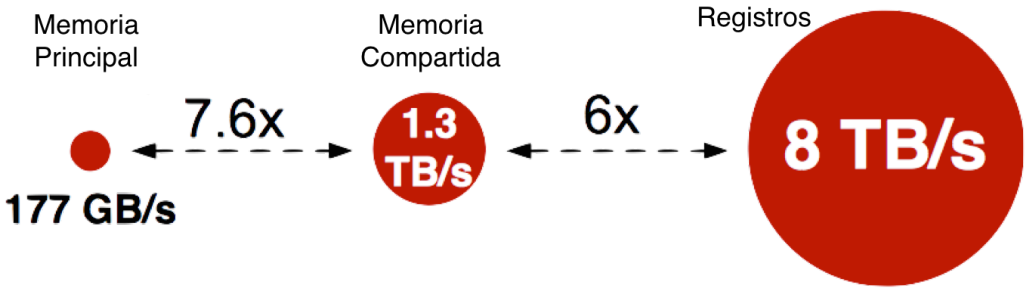


Fig. 8 - Comparativa de la velocidad de los tipos de memoria[7].

En conclusión, la GPU es capaz de procesar muchos datos en poco tiempo, en comparación, la memoria principal es muy lenta; por lo que su principal cuello de botella es el acceso a los datos; los registros son muy rápidos, pero usar demasiados implica que el Kernel dispone de un menor numero de hilos para trabajar. El programador deberá decidir en cada momento según la mecánica del problema que le supone mas ventajoso. Una tarea donde se realizan gran cantidad de operaciones con pocos accesos a memoria se vera beneficiado una mayor cantidad de hilos con los que trabajar; mientras que, por el contrario, en una tarea donde se realizan muchos accesos a memoria, probablemente, compense renunciar a un mayor numero hilos en favor de un uso mayor del numero de registros.

Debido a que es necesario un conocimiento profundo para poder realizar cualquier proyecto, han surgido alternativas que tratan de abstraer al programador de la arquitectura y de toda la complejidad que conlleva el uso de CUDA para poder realizar traslados de un código CPU fácilmente a las GPU. Una de esas alternativas es OpenACC [11] del cual hablare mas adelante.

3.2.4. OpenACC ¿Es realmente una alternativa?

Durante la elaboración del proyecto surgió la posibilidad de usar una interfaz llamada OpenACC similar a la usada en OpenMP para la creación de Kernels CUDA mediante el uso de directivas de compilación, por lo que decidí realizar unas pruebas con el clásico problema de multiplicación de matrices que ya hemos visto en el punto dedicado a OpenMP.

Código Multiplicación de matrices en OpenACC

```
void  multiplicaMatriz(int numColumnas1, int numFilas1, int numColumnas2, int
numFilas2, double *restrict m1, double *restrict m2, double *restrict resultado) {
   #pragma acc kernels
copyin(m1[0:numFilas1*numColumnas1],m2[0:numFilas2*numColumnas2]),
copyout(resultado[0:numFilas1*numFilas2])
   {
      int i;
      #pragma acc loop gang vector(256), independent
      for (i=0;i<numFilas1;i++)
      {
         int j;
         #pragma acc loop gang vector(2) independent
         for(j=0;j<numFilas2;j++)
         {
            double acumulador = 0;
            int k;
            for(k=0;k<numColumnas1;k++)
               acumulador += m1[i+k*numFilas1] * m2[j*numFilas2+k];
            resultado[i*numFilas1+j] = acumulador;
         }
      }
   }
}
```

Código Multiplicación de matrices en CUDA

```
#define TILE_SIZE 32
                                    ..................
                                    ..................

   //Reservando memoria en la GPU
   cudaMallocPitch(&m1_d, &pitch1,numColumnas1*sizeof(real_t),numFilas1);
   cudaMallocPitch(&m2_d, &pitch2,numColumnas2*sizeof(real_t),numFilas2);
   cudaMallocPitch(&m3_d, &pitch3,numColumnas2*sizeof(real_t),numFilas1);
   cudaMemcpy2D(m1_d, pitch1, m1,
numColumnas1*sizeof(double),numColumnas1*sizeof(double),numFilas1,
cudaMemcpyHostToDevice);
   cudaMemcpy2D(m2_d, pitch2, m2, numColumnas2*sizeof(double),
numColumnas2*sizeof(double),numFilas2, cudaMemcpyHostToDevice);

   //Planificacion de hilos
   int n_blocks_x = numColumnas2/TILE_SIZE + (numColumnas2%TILE_SIZE == 0
? 0:1);
   int n_blocks_y = numFilas1/TILE_SIZE + (numFilas1%TILE_SIZE == 0 ? 0:1);
```

```
    dim3 block_size_dim(TILE_SIZE,TILE_SIZE);
    dim3 n_blocks_dim(n_blocks_x,n_blocks_y);

    int ipr1 = pitch1/sizeof(double);
    int ipr2 = pitch2/sizeof(double);
    int ipr3 = pitch3/sizeof(double);

    //Invocacion del Kernel, con la estructura de hilos requerida
    multMatrizCuda <<< n_blocks_dim, block_size_dim >>> (m1_d,m2_d,m3_d,
numFilas1, numColumnas2, numFilas2,ipr1,ipr2,ipr3);
    cudaMemcpy2D(m3,numColumnas2*sizeof(real_t),m3_d,pitch3,
numColumnas2*sizeof(real_t), numFilas1, cudaMemcpyDeviceToHost);

                                        ……
                                        ……

__global__ void multMatrizCuda(double* m1, double* m2, double* m3, int
numFilas1, int numColumnas2, int profundidad, int ipr1, int ipr2, int ipr3) {
    //Cordenadas de cada hilo a partir del cual se asigna el trabajo
    int j  = blockIdx.x * blockDim.x + threadIdx.x;
    int i  = blockIdx.y * blockDim.y + threadIdx.y;
    double ac = 0;

    __shared__ real_t trozo_m1[TILE_SIZE][TILE_SIZE];
    __shared__ real_t trozo_m2[TILE_SIZE][TILE_SIZE];

    for (int elements_done = 0; elements_done<profundidad;elements_done +=
blockDim.x ) {
        trozo_m1[threadIdx.y][threadIdx.x]= m1[i*ipr1 + elements_done + threadIdx.x];
        trozo_m2[threadIdx.y][threadIdx.x]= m2[(elements_done + threadIdx.y)*ipr2+j];

        __syncthreads();

        int k_limit = min(blockDim.x, profundidad-elements_done);

        for(int k=0; k<k_limit ;k++)
        ac += trozo_m1[threadIdx.y][k] * trozo_m2[k][threadIdx.x];
    }
    if((j<numColumnas2) && (i<numFilas1)) m3[i*ipr3+j] = ac;
}
```

Salta a la vista es la cantidad de código cada hace falta en CUDA respecto a la versión OpenACC para hacer la misma tarea, OpenACC simplifica la reserva de memoria en la GPU por medio de directivas de forma similar a como se haría en OpenMP para declarar que una variable es compartida o privada; además en la versión

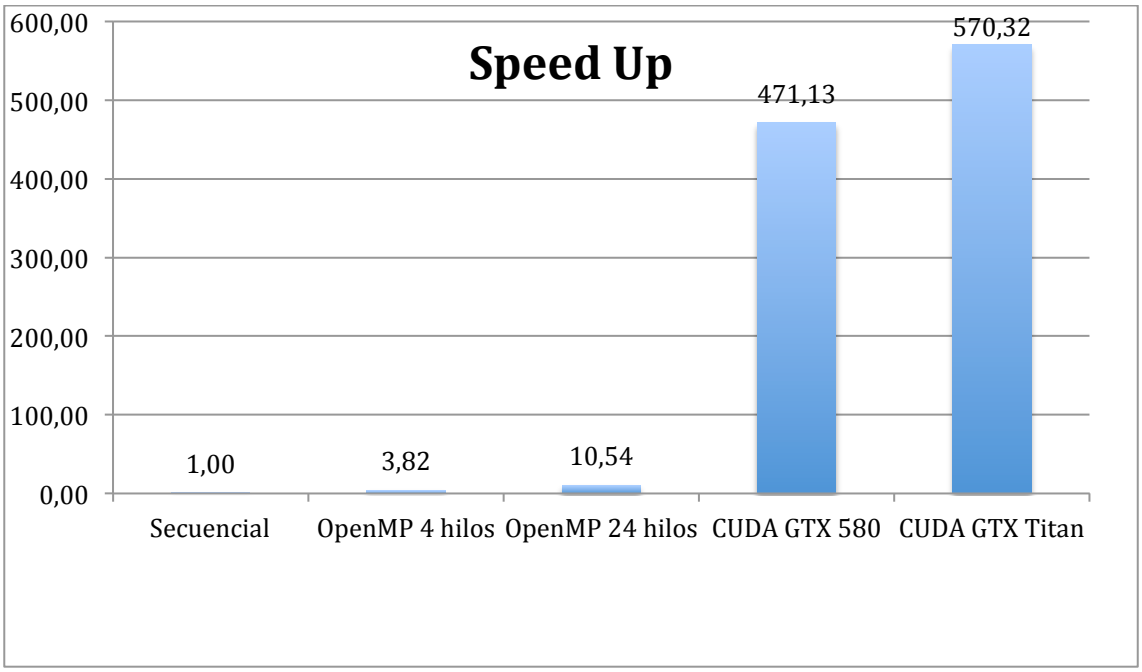
OpenACC el programador puede abstraerse del reparto de hilos, pues OpenACC se ocupa también de ello.

He de mencionar en este punto, que actualmente OpenACC solo es soportado por unos pocos compiladores propietarios, por lo que los experimentos que pude hacer se hicieron mediante el uso de las versiones de prueba de uno de los compiladores.

Aunque en un primer momento la filosofía de OpenACC, que promete abstraer al programador de toda la complejidad detrás del uso de GPUs, parece ideal; en su estado actual no sirve para la mayoría de los proyectos logrando un rendimiento pobre en comparación con el de un programa completamente adaptado mediante CUDA, aunque podría resultar útil, en una primera etapa de la adaptación, para ver si el problema se puede adecuar bien a la arquitectura.

3.3. COMPARATIVA DE TECNOLOGÍAS DE PARALELIZACION

El potencial de CUDA como dispositivo paralelismo masivo es muy prometedor, lo que hace que valga la pena el esfuerzo extra necesario para exprimirlo al maximo.



Graf. 2 - Comparativa de rendimiento del problema de multiplicación de matrices usando distintas tecnologías para matrices cuadradas de tamaño 3000.

4. DESARROLLO DEL PROYECTO

A continuación voy a relatar los pasos seguidos durante el desarrollo del proyecto para la optimización y paralelización del algoritmo SL. El desarrollo esta dividido en tres etapas diferenciadas y por ello voy a relatarlas por separado.

Estas etapas son la de optimización secuencial del algoritmo, donde explicare varias de las soluciones propuestas y los resultados obtenidos; la etapa paralelización mediante la tecnología de OpenMP, en la cual hablare de cómo he implementado este método de paralelización, que problemas han surgido y como he podido resolverlos; por ultimo, hablare de mi acercamiento a la tecnología Nvidia CUDA para el uso de GPUs como dispositivos de co-procesamiento de datos.

4.1. MEJORA DEL PROGRAMA SECUENCIAL

4.1.1. ANÁLISIS DE RENDIMIENTO

Como medida inicial de este proyecto se hizo una fiel adaptación al código de programación C, desde una versión anterior escrita en lenguaje matemático *Matlab*, por ser un lenguaje rápido y que permite el manejo de memoria a bajo nivel lo que nos da más juego a la hora de optimizar el código; tras lo cual se realizo un profundo análisis de rendimiento del programa, mediante la herramienta de depuración *Valgrind*, para encontrar aquellas ineficiencias en el programa que pudieran ser corregidas o eliminadas.

A continuación se explican las principales ideas surgidas del análisis para la mejora del programa.

4.1.1.1 UNIFICACIÓN DE BUCLES

A la hora de optimizar un programa es importante evitar abusar de los saltos condicionales, pues consumen muchos ciclos. Por ello, si vemos que varios bucles son similares hay que pensar si no es posible que pueda realizarse la misma tarea usando un único bucle.

En la implementación preliminar hay varios bucles que podrían ser unificados sin mucha dificultad.

Como ya se ha descrito en un anteriormente, el algoritmo se dividen en dos fases, la primera se encarga de hacer el análisis de distancias en cada canal y la segunda se encarga de comparar el resultado entre todos los canales. Tal como estaba programado en Matlab, y luego en nuestra versión preliminar, la primera fase se realiza para todos los tiempos almacenando los resultados, y luego se realizaba la segunda fase de nuevo para todos los tiempos.

Versión preliminar:
*Para cada **i** perteneciente a **I**:* *Calcular distancias* *Para cada **i** perteneciente a **I**:* *Comparar canales*

Tras analizar el programa, quedo claro que podría unificarse ambos bucles, permitiendo no solo reducir a la mitad el numero de saltos, sino también ahorrar memoria al no tener que almacenar los resultados del análisis de todos los tiempos en la primera fase del algoritmo.

Solución propuesta:
*Para cada **i** perteneciente a **I**:* *Calcular distancias* *Comparar canales*

En la fase de análisis de un canal se realiza una búsqueda hacia atrás y otra hacia delante en el tiempo, respecto a la señal del canal que se esta analizando, por medio de dos bucles; ambos tienen los mismos rangos de iteración, salvo que uno suma el valor del iterador al instante presente y el otro lo resta.

Estructura actual:
*Para cada **i** perteneciente a **I**:* *Para cada **w** perteneciente a **W**:* *Calcular distancia(**i**, +**w**);* *Para cada **w** perteneciente a **W**:* *Calcular distancia(**i**, -**w**);*

Por lo que no seria complicado unificar ambos bucles haciendo que en uno se realizaran ambas tareas.

Estructura sugerida:
*Para cada **i** perteneciente a **I**:* *Para cada **w** perteneciente a **W**:* *Calcular distancia(**i**, +**w**);* *Calcular distancia(**i**, -**w**);*

4.1.1.2. SUPRESIÓN DE OPERADORES INNECESARIOS

Para poder quedarnos con los *N* elementos mas próximos al instante analizado, calculamos la distancia euclídea entre el instante presente y el resto de tiempos dentro de la ventana *W*. Por lo tanto estamos realizando la operación raíz cuadrada, operación muy costosa computacionalmente hablando, hasta miles de veces según el tamaño de las pruebas.

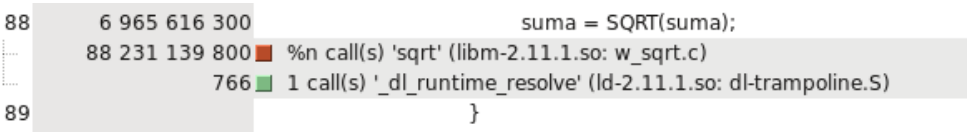


Fig. 9 - Imagen extraída de Kcachegrind

Siendo que para el caso no es necesario conocer el valor, sino mas concretamente su relación con los demás, podríamos obviar esa operación.

$$\leq \frac{\sqrt[2]{(p_{11}-q_{11})^2+(p_{12}-q_{12})^2+\cdots+(p_{1n}+q_{1n})^2}}{\sqrt[2]{(p_{21}-q_{21})^2+(p_{22}-q_{22})^2+\cdots+(p_{2n}+q_{2n})^2}}$$

$$\Updownarrow$$

$$\leq \frac{(p_{11}-q_{11})^2+(p_{12}-q_{12})^2+\cdots+(p_{1n}+q_{1n})^2}{(p_{21}-q_{21})^2+(p_{22}-q_{22})^2+\cdots+(p_{2n}+q_{2n})^2}$$

4.1.1.3. ALMACENAMIENTO DE CÁLCULOS REPETITIVOS

Por como esta definida la ventana de tiempos *W* sobre la que se realiza el análisis, realmente cada cálculo de distancia se hacía dos veces; una primera vez al calcular la distancia hacia el futuro, y una segunda al calcular hacia el pasado.

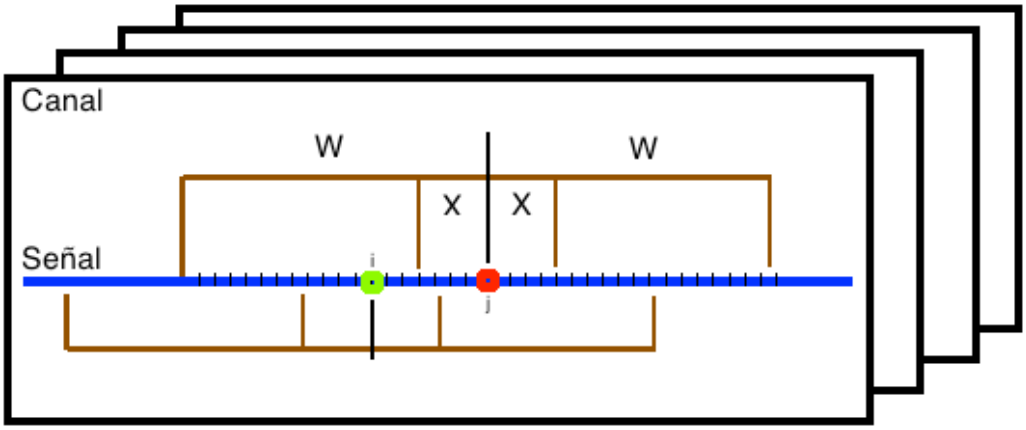


Fig. 10 - Las ventanas de dos instantes de tiempo pueden superponerse y por tanto repetir cálculos de forma innecesaria.

Se considero que podría ser interesante tratar de almacenar la distancia entre *i* y *j* tras calcularlo por primera vez para poder reutilizarlo al ir a calcular la distancia entre *j* e *i*.

4.1.1.4. DISTINCIÓN DE CASOS EXTREMOS

Como ya he explicado para el caso de los bucles los saltos condicionales consumen muchos ciclos; por lo que si pueden eliminarse, o evitar que se repitan infinidad de veces, puede lograrse una gran ganancia de rendimiento en un programa.

Un claro ejemplo de saltos condicionales evitables es el que encontramos en la consideración de los casos limítrofes del algoritmo.

El algoritmo SL consume datos en una amplia ventana de tiempos dentro de una señal, lo cual implica que en los extremos al principio y al final hay que considerarlo de forma especial. Esta tarea la realiza la versión preliminar mediante el uso de la estructura *if-else* durante toda la ejecución del programa, por lo cual se están realizando comprobaciones innecesarias durante gran parte de la ejecución.

Una mejor aproximación podría realizarse mediante la separación de la ejecución del algoritmo en dos partes. El caso de los extremos, con la estructura *if-else*; y el caso interior, sin ningún tipo de consideración por los casos extremos. Dejando que un nivel superior del programa se encargue de tomar la decisión de sobre que tiempos se ha de ejecutar cada uno de los casos.

4.1.1.5. COMPRESIÓN DE LAS ACCESOS A DATOS

Cuando un procesador accede a memoria, éste se lleva un bloque entero desde la memoria principal a la, mas rápida, memoria cache. Para una optima utilización de la memoria cache, es recomendable no ocupar memoria inútilmente; esto quiere decir, que si puedes contar una cosa con una palabra, entonces no uses diez.

En la implementación preliminar se almacenan los *N* tiempos más próximos a la señal analizada en la primera fase del algoritmo. Almacenando los *N* próximos en un vector de tamaño W, rellenado con *0*, como *1* en su posición con respecto a la ventana *W*; de esta forma en la fase dos sólo hay que comparar los vectores de dos canales para hallar el numero de coincidencias.

Esto, quizá podría hacerse de forma más eficiente si los vectores fueran de tamaño N y en lugar de 0 o 1 se almacenase su posición en la ventana W; de tal forma que con que los valores del vector coincidieran con los de otro seria suficiente.

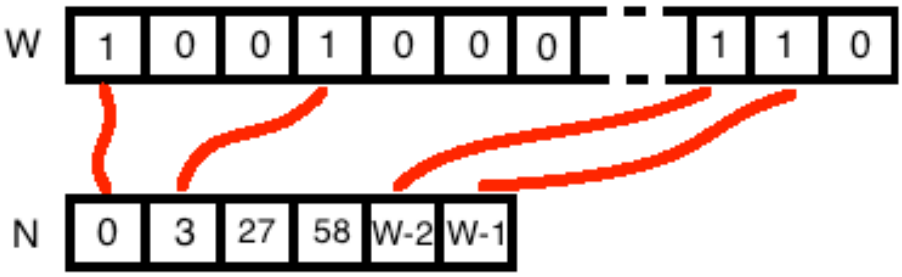


*Fig. 11 - Vector de tamaño **W**, con valores booleanos, y su equivalente en un vector de tamaño **N**.*

4.1.1.6. APROVECHAMIENTO DE LA SIMETRIA

En la segunda fase del algoritmo se compara cada canal con los demás, esto provoca que cada canal se compare dos veces con los demás produciendo como resultado una matriz simétrica con diagonal 1.

$$\begin{pmatrix} 0,0 & \cdots & 0,k \\ \vdots & \ddots & \vdots \\ k,0 & \cdots & k,k \end{pmatrix}$$

De nuevo estamos haciendo que el programa trabaje de más, cuando podríamos hallar el mismo resultado haciendo que no se comparara con canales ya utilizados.

4.1.2. OPTIMIZACIÓN

Una vez encontrados e identificado los puntos de mayor consumo de tiempo e ideado métodos para mejorar el rendimiento del programa, es el momento de ponerlos en práctica e implementar los cambios, en esta fase aún no se ha realizado ningún tipo de paralelización, por lo que toda mejora es a nivel secuencial.

Para poder sacar un buen rendimiento debemos tener en cuenta una serie de buenas practicas, no abusar de funciones costosas, reduje el número de saltos condicionales, etc.

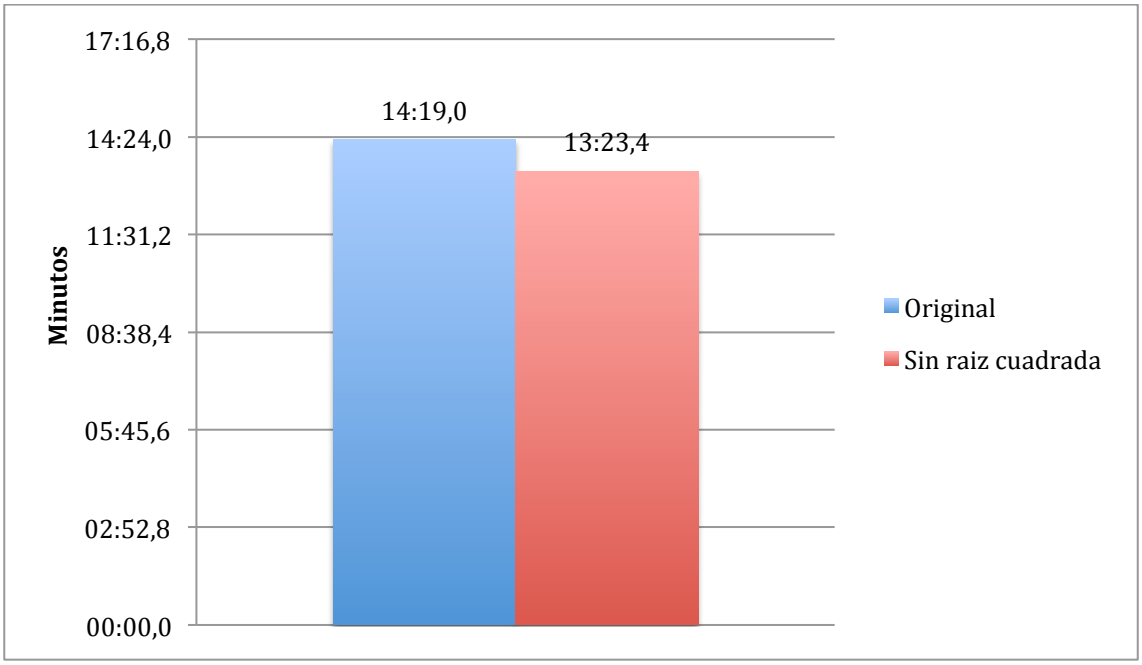
Ahora voy a relatar los cambios mas destacables y comparar su rendimiento con un experimento base de la implementación preliminar con unos datos de entrada a los que me referiré como Paciente_test con 148 canales y 50863 muestras de tiempo en cada uno.

Todas las pruebas se han realizado en la maquina Magnolio referida en el ANEXO.

4.1.2.1. SUPRIMIENDO OPERADORES INNECESARIOS

En la implementación preliminar del programa al tratar de ser lo mas fiel posible a la definición del algoritmo hay varias operaciones que aunque válidas, son computacionalmente costosas e innecesarias, como el caso ya citado del uso de la raíz cuadrada en el cálculo de las distancias euclídeas.

En la siguiente gráfica veremos la mejora de rendimiento obtenida tras su eliminación.



Graf. 3 - Comparación entre el experimento base y el experimento eliminando el operador de raíz cuadrada.

4.1.2.2 ALMACENANDO DE CÁLCULO DE DISTANCIAS

Otra de las mejoras propuestas, era almacenar el calculo de las distancias realizadas en la fase de análisis de un canal, pues ocurría que las ventanas de tiempo *W* se solapaban y eso provocaba la repetición del cálculo de la distancia euclídea entre dos puntos. Por ello, se optó por almacenar en una tabla en memoria el primer calculo y limitarnos a leer de la tabla la segunda vez.

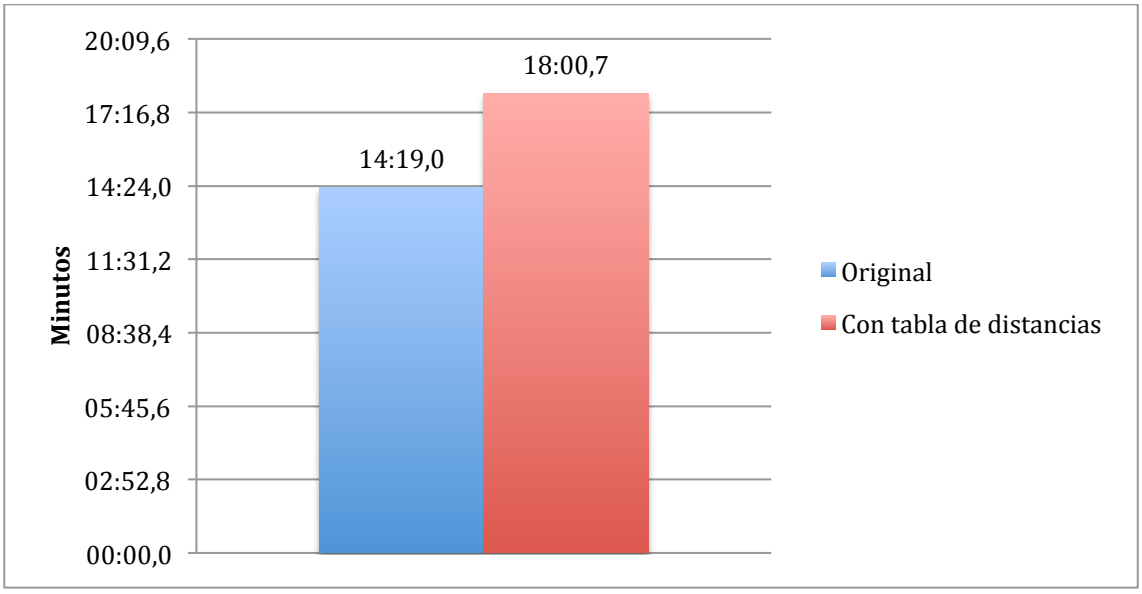
Para ello primero se ideo la siguiente tabla.

x/y	0	1	2	3	4	5	6	7	8	9	10	11	12	13
13											13,17	13,18	13,19	13,13
12										12,16	12,17	12,18	12,12	
11									11,15	11,16	11,17	11,11		
10								10,14	10,15	10,16	10,10			
9							9,13	9,14	9,15	9,9				9,13
8						8,12	8,13	8,14	8,8				8,12	8,13
7					7,11	7,12	7,13	7,7				7,11	7,12	7,13
6				6,10	6,11	6,12	6,6				6,10	6,11	6,12	
5			5,9	5,10	5,11	5,5				5,9	5,10	5,11		
4		4,8	4,9	4,10	4,4				4,8	4,9	4,10			
3	3,7	3,8	3,9	3,3				3,7	3,8	3,9				
2	2,7	2,8	2,2				2,6	2,7	2,8					
1	1,7	1,1				1,5	1,6	1,7						
0	0,0				0,4	0,5	0,6							

*Fig. 12 - Tabla de distancias de tamaño **W** x **W**.*

La tabla es una matriz cuadrada que representa en ambas dimensiones el tamaño de un lado de la ventana, que hará a veces de futuro y otras de pasado; cuando calculamos las distancias entre el instante actual y su futuro, se guardan los cálculos en la tabla en su respectiva posición. De esta forma, cuando tengamos que calcular la distancia desde el futuro al pasado ya tendremos los cálculos hecho y tan solo habrá que acceder a la posición correspondiente de la tabla.

Pero lamentablemente al ser una tabla grande y estar los datos dispersos en memoria, éste cambio resultó ser poco efectivo e incluso perjudicaba a la duración de los tiempos de ejecución del programa.



Graf. 4 - Comparación de tiempos entre el experimento base y un experimento con tabla de distancias.

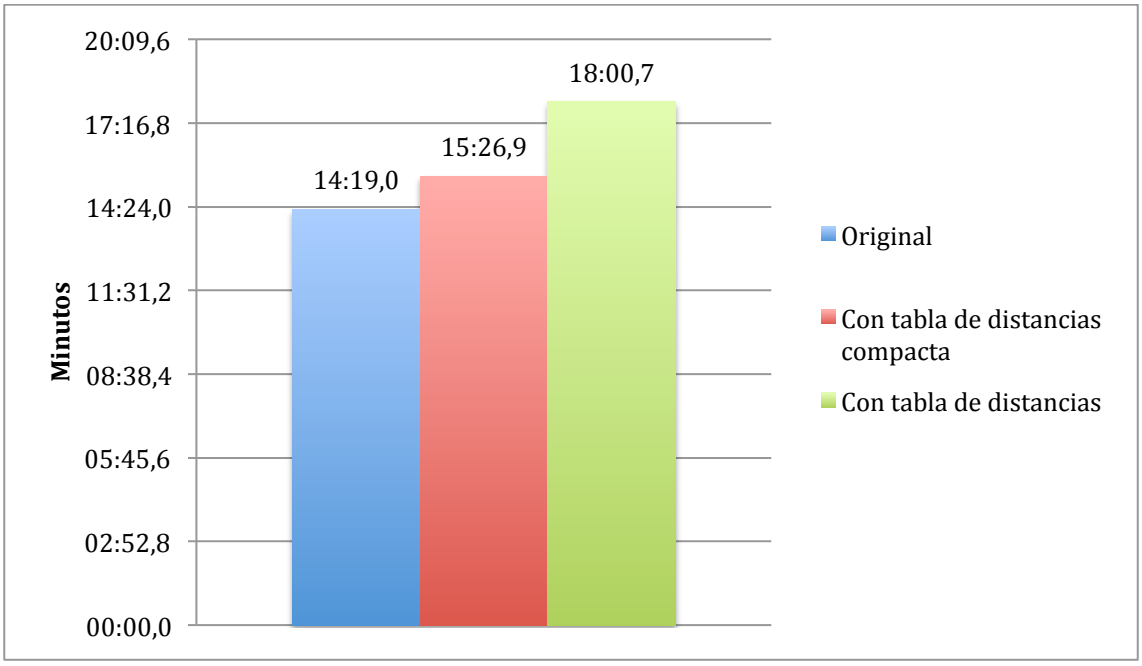
Viendo el pobre rendimiento de la primera tabla se decidió rediseñarla por una más compacta de forma que los datos estuvieran mejor alineados en memoria y fuera de más ágil acceso.

Primero reducimos el tamaño de la tabla acotándola al máximo al problema, eliminando las posiciones que por la lógica del algoritmo no iban a ser accedidas. Luego mejoramos el cálculo de los índices de acceso de forma que los accesos a escritura, los marcos en azul en la tabla, se hicieran en lugares contiguos en memoria.

x y				
	13	13,17	13,18	13,19
	12	12,16	12,17	12,18
	11	11,15	11,16	11,17
	10	10,14	10,15	10,16
	9	9,13	9,14	9,15
	8	8,12	8,13	8,14
	7	7,11	7,12	7,13
	6	6,1	6,11	6,12
	5	5,9	5,1	5,11
	4	4,8	4,9	4,1
	3	3,7	3,8	3,9
	2	2,6	2,7	2,8
	1	1,5	1,6	1,7
	0	0,4	0,5	0,6
		0	1	2
		z		

Fig. 13 - Tabla de distancias compacta.

Con esta versión se logro una mejora sustancial con respecto al a primera, pero no conseguí reducir los tiempos con respecto a la versión sin tabla.



Graf. 5 - Comparativa entre el experimento base y experimentos con las dos versiones de tablas.

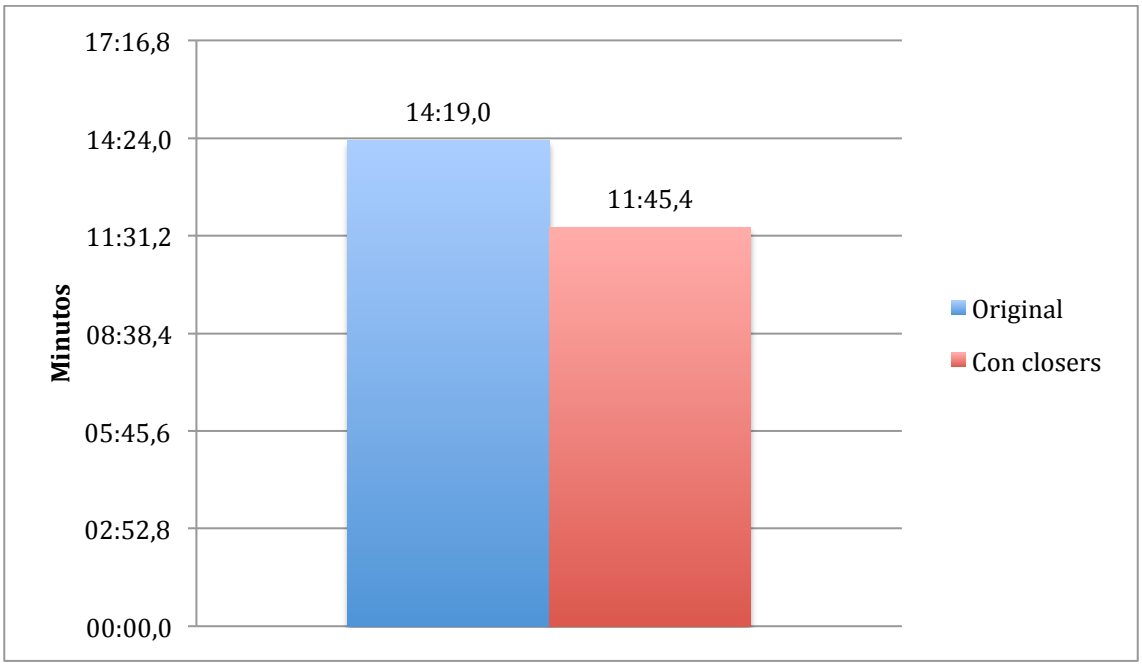
Finalmente este cambio tuvo que ser descartado, porque aunque la idea parecía prometedora, como puede apreciarse en la gráfica, el ahorro en los cálculos no consigue compensar los tiempos de acceso a memoria.

4.1.2.3 ALMACENADO DE N PRÓXIMOS

Para realizar la mejora explicada en el punto cinco del apartado de análisis del programa secuencial, creé un vector de tamaño *N*, al que me referiré como vector de *closers*, donde fui almacenando la posición relativa a la ventana *W* de cada valor, siendo descartadas las posiciones más antiguas a medida que iban surgiendo nuevas posiciones más próximas, hasta completar la ventana; de forma que al termino del análisis de una ventana solo quedan en el vector de *closers* las posiciones de los instantes cuyo valor es más próximos al del punto de origen o presente de la ventana.

En la segunda fase, para cada canal *k*, creo un vector de tamaño *W* al que llamare *intermedio*, y fijamos las *N* posiciones del vector *closers* a 1; luego sumamos los valores obtenidos por medio de usar el vector *closers* del canal *h* como índices para el vector *intermedio*, lo que nos da como resultado el numero de coincidencias entre el canal k y el canal h.

En la siguiente gráfica puede apreciarse una significativa mejoría en los tiempos de ejecución del programa tras realizar esta modificación.



*Graf. 6 - Comparativa entre el experimento base y el experimento con **closers**.*

4.1.2.4 APROVECHANDO LA SIMETRÍA

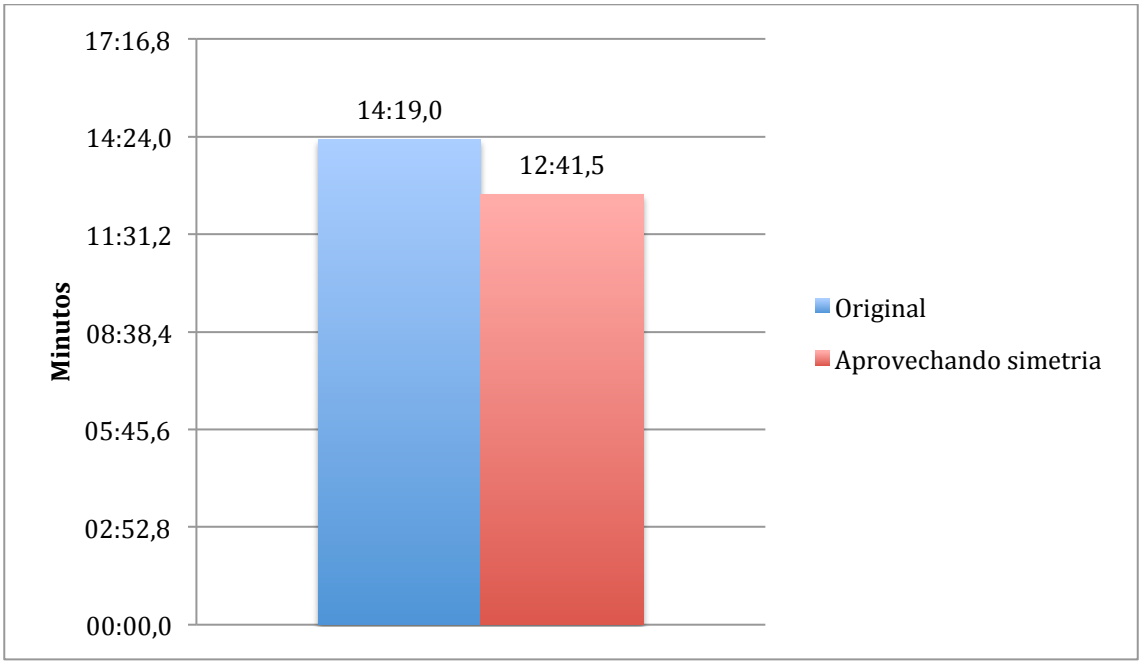
Como he dejado claro en anteriormente, la segunda fase del algoritmo es una operación simétrica; es por ello lógico pensar que si nos aprovechamos de su simetría, y en lugar de comparar cada canal con todos los demás, comparamos tan solo cada canal con los siguientes deberíamos notar una mejora bastante elevada en el rendimiento del programa. Luego solo hay que almacenar el resultado de la comparación en los dos lugares de la matriz que le corresponde.

$$M(i,j) = M(j,i)$$

También hay que tener en cuenta que la comparación de un canal consigo mismo es innecesaria por que siempre va a coincidir al 100%, es por ello que también podemos ahorrarnos ese trabajo.

$$M(i,i) = 1$$

A continuación veremos una gráfica presentando la comparación de tiempos antes y después de eliminar la redundancia de cálculos de la segunda fase del algoritmo.



Graf. 7 - Comparativa del experimento base y un experimento aprovechando la simetría de la segunda fase del algoritmo.

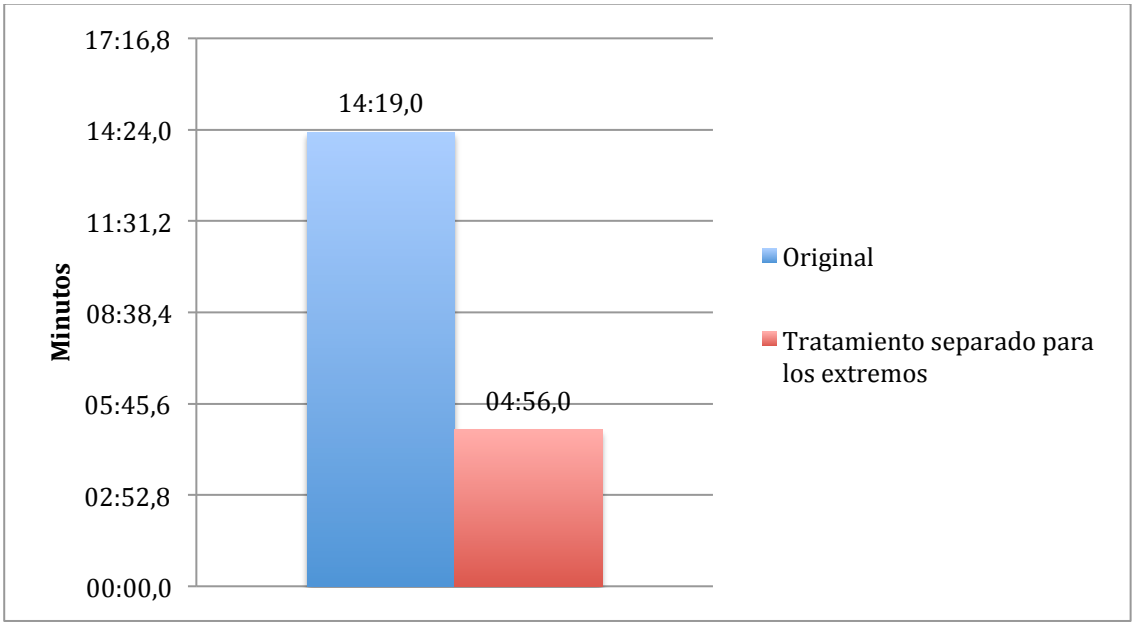
Podemos observar, que habiendo reducido a la mitad el trabajo de la segunda fase del algoritmo la mejora del rendimiento no refleja esa misma proporción, esto se debe a que la primera fase del algoritmo acapara mucho más tiempo de ejecución. A pesar de ello si que se puede percibir una mejora, por ello se decidió conservar la modificación en la implementación.

4.1.2.5 Distinción de la ejecución en los casos extremos

Hasta ahora no hemos visto mejoras radicales en el rendimiento. Esto es debido a que, a excepción del fallido intento de la tabla de distancias, los cambios realizados se han centrado principalmente en la mejora de la segunda fase del algoritmo pero realmente lo que más tiempo lleva es la primera fase; por lo que para poder tener una ganancia sustancial tendremos que optimizar ésta.

Ya he explicado antes que los saltos condicionales, y mas si estos están dentro de algún tipo de recursión de magnitud elevada, son instrucciones que consumen muchos ciclos del procesador; por lo que se decidió eliminar el tratamiento de condiciones especiales del principal cuerpo de ejecución y contemplarlos en un nivel superior del programa que se encargue de tomar la decisión de que ejecutar en cada momento.

Para ello he realizado dos implementaciones del algoritmo, una para esos casos límite al principio y al final del análisis de datos que requieren de consideraciones especiales y otra para el análisis de los datos centrales, los cuales ocupan el mayor tiempo en la ejecución, en la que no se tiene en cuenta ningún tipo de condición; pues ya se suponen consideradas antes de su llamada.



Graf. 8 - Comparativa entre el experimento base y un experimento en el cual tratamos de forma separada los casos extremos.

En la grafica se observa una ganancia considerable con respecto a la versión preliminar. Debido a que al no tener que estar haciendo comprobaciones innecesarias durante la parte interior, el cual en una ejecución normal se llevara todo el tiempo, el programa ahorra muchos ciclos.

4.1.2.6 ALINEAMIENTO DE DATOS

Los datos almacenados en memoria se han mantenido en el mismo formato en el que se leía del fichero de entrada; esto es, instantes de tiempo alineados en filas y canales en columnas.

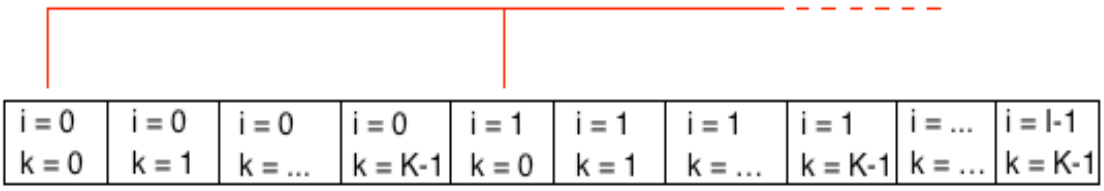


Fig. 14 - Acceso disperso a un vector de datos.

Como resultado de ello, dos accesos consecutivos en ejecución y pertenecientes al mismo canal acceden a dos regiones completamente diferentes de memoria, provocando un pobre aprovechamiento de las caches.

Para solucionar esto, vamos ha almacenar los datos de forma inversa; los instantes de tiempo estarán en columnas y los canales en filas. De manera que en memoria quedaría como puede verse en la siguiente figura.

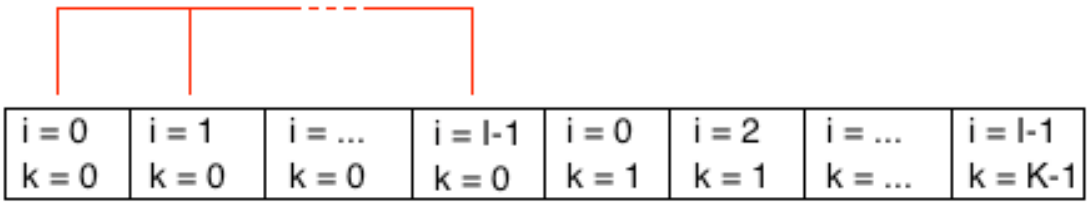
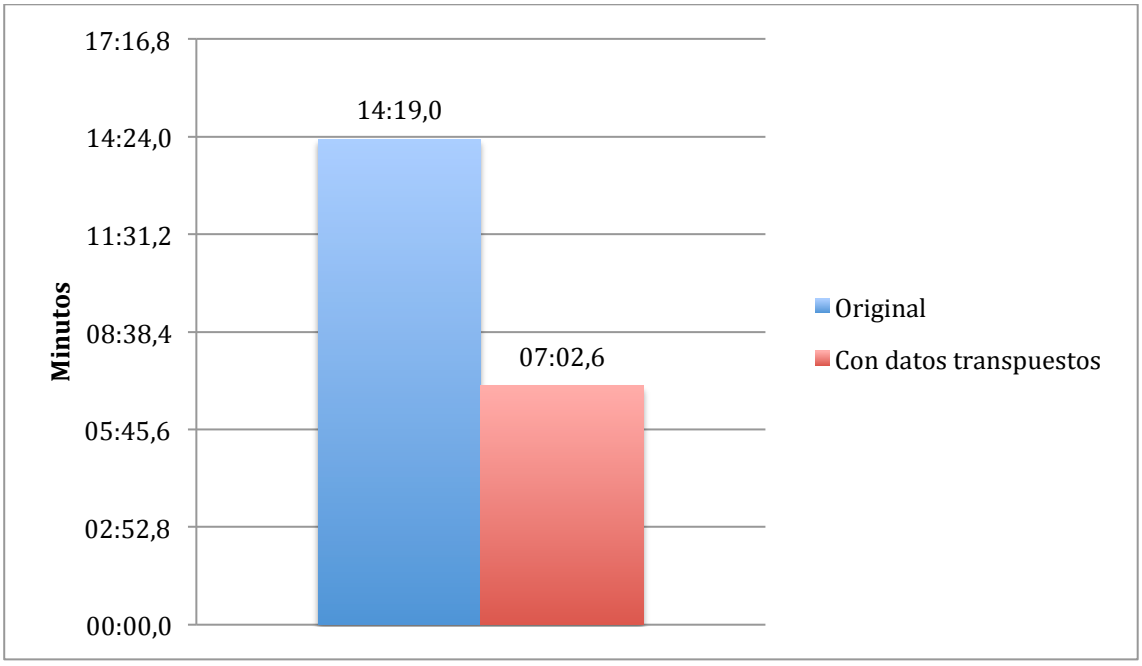


Fig. 15 - Acceso alineado a un vector de datos.

A continuación una grafica con los resultados obtenidos tras el cambio.

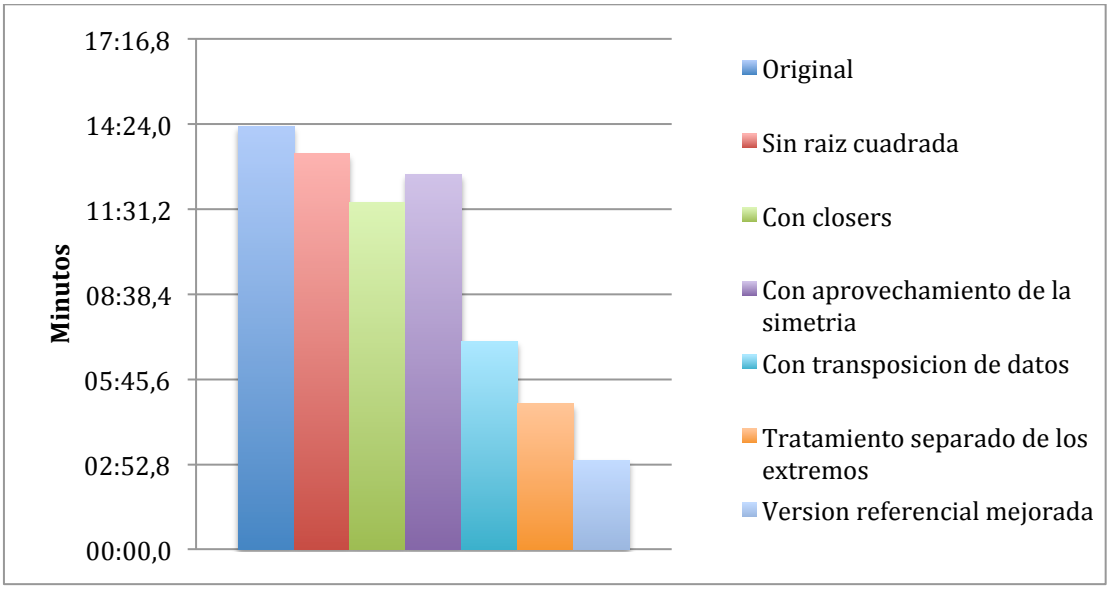


Graf. 9 - Comparativa entre el experimento base y un experimento con transposición de datos.

Podemos ver que el transponer los datos ha beneficiado enormemente al tiempo de ejecución, esto es debido al aprovechamiento de la ***proximidad referencial***; Al leer un dato se sube a la cache del procesador un bloque entero de memoria, por lo que si accedemos a regiones contiguas, nos ahorramos lecturas de memoria principal al encontrarse de forma anticipada los datos en la memoria cache del procesador.

4.1.3 Unificación y comparativa de las mejoras

Finalmente vamos a ver una grafica comparativa entre la versión original, cada intento de mejora satisfactoria y finalmente una ejecución en la cual hemos integrado todas los cambios que han producido una mejoría en el rendimiento del programa.



Graf. 10 - Comparativa completa de los experimentos.

4.2 PARALELIZACION CON OPENMP

Habiendo quedado satisfecho con el rendimiento de la implementación secuencial del algoritmo SL, pasamos a buscar los puntos más adecuados para su paralelización.

La tecnología que he usado para paralelizar el algoritmo ha sido OpenMP, que como ya se he explicado anteriormente proporciona una interfaz sencilla para la programación multiproceso de memoria compartida.

Para las pruebas se realizaran los experimentos en la maquina Magnolio referida en el ANEXO.

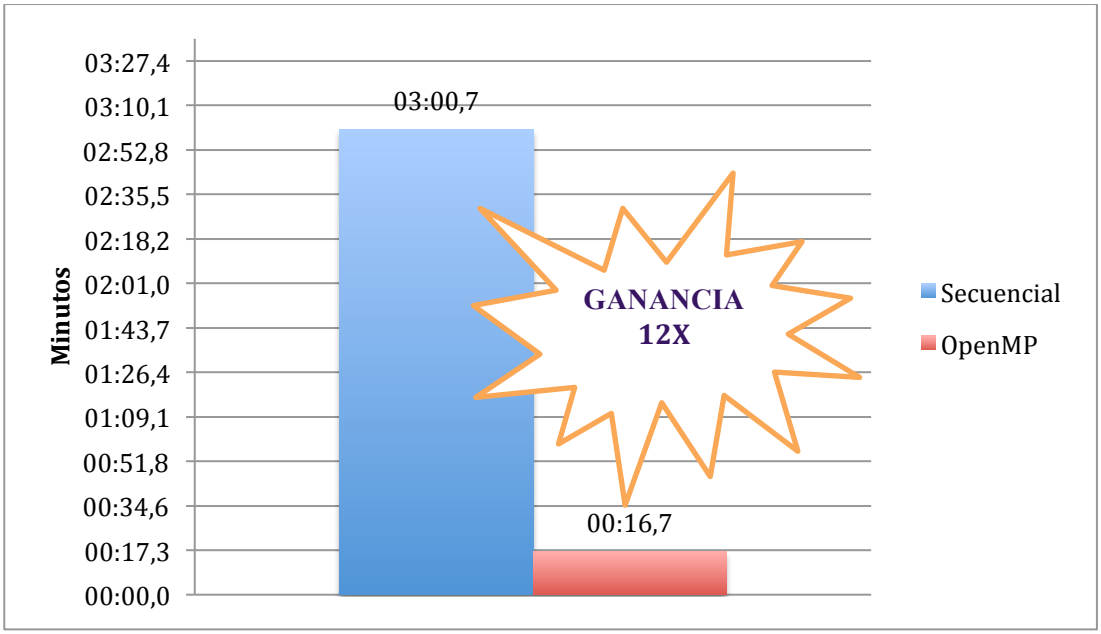
4.2.1. PRIMER CERCAMIENTO

Para conseguir un buen desempeño de la paralelización se ha decidido declarar como región paralela el bucle mas externo, que se encarga de recorrer cada uno de los instantes de tiempo I de los datos de entrada.

Codigo directivas OpenMP

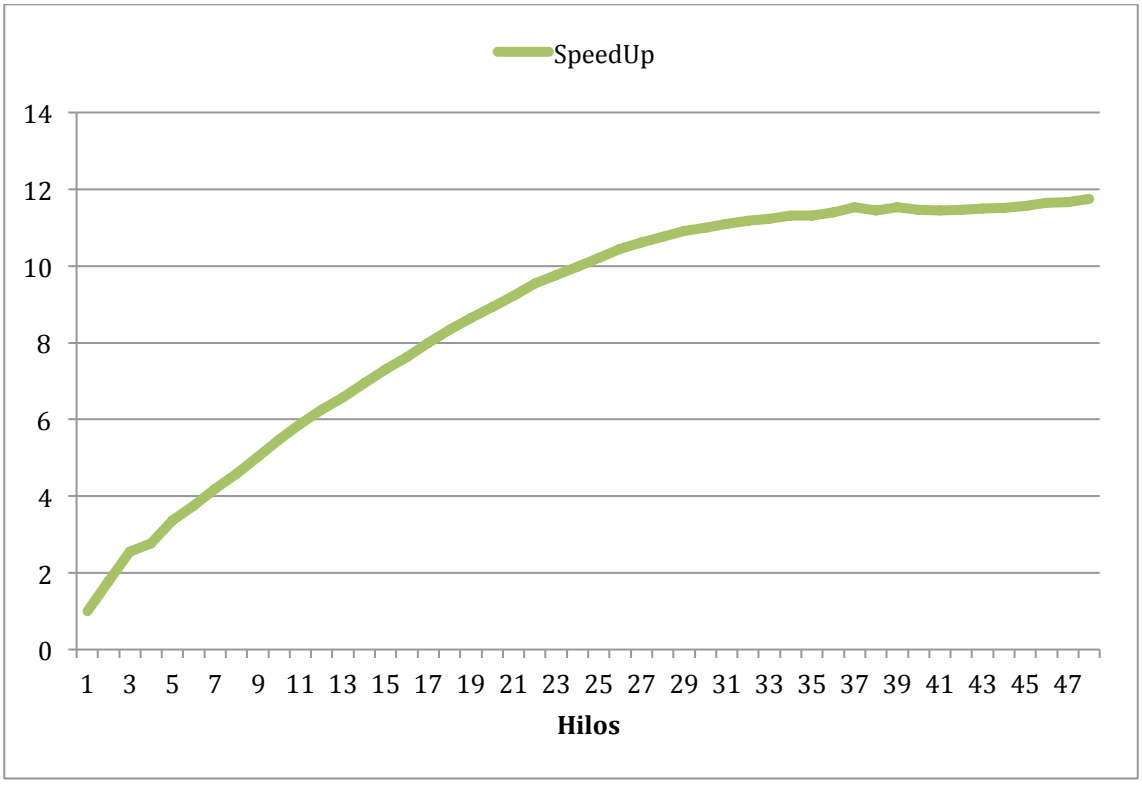
```
#pragma omp parallel for default(none) private(i) shared(sync_like,data,stderr,ret)
for (i=Ib; i<Ie; i+=S)
{
                                        ...
}
```

A continuación veremos una grafica con la comparativa de tiempos entre la versión secuencial y la versión multi-hilo mediante OpenMP.



Graf. 11 - Comparativa entre la versión referencial mejorada y la versión paralela usando 48 hilos.

Teniendo en cuenta que se usan 48 hilos, una ganancia de 12 en el tiempo de ejecución no me parecía demasiado prometedor; por ello realice un estudio para ver como se comportaba el programa y la degradación de la ganancia según aumentaba el numero de hilos.



Graf. 12 - Comparativa de la ganancia al aumentar el numero de hilos.

4.2.2. PROBLEMAS

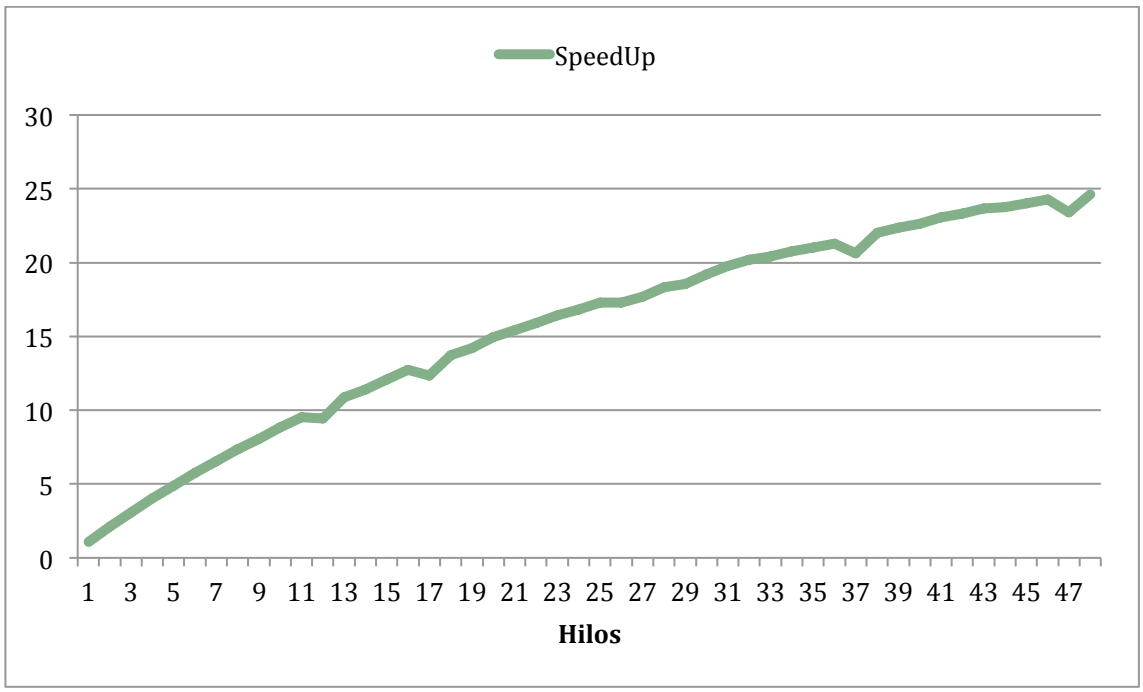
En este punto, dada la rápida degradación en la ganancia observada al ir aumentando el numero de hilos, estaba claro que había algún tipo de problema que provocaba que los hilos se estorbaran los unos a los otros.

Al repasar el código y realizar varias pruebas, finalmente se concluyo que el problema se debía a que los hilos, a pesar de que cada hilo escribía sobre variables diferentes, al trabajar sobre los mismos bloque, perdían mucho tiempo compitiendo por el acceso a los mismos bloques de memoria, con la consiguiente anulación constante de las caches de cada procesador. A este problema se le conoce como ***falsa compartición***.

4.2.3. SOLUCIONES

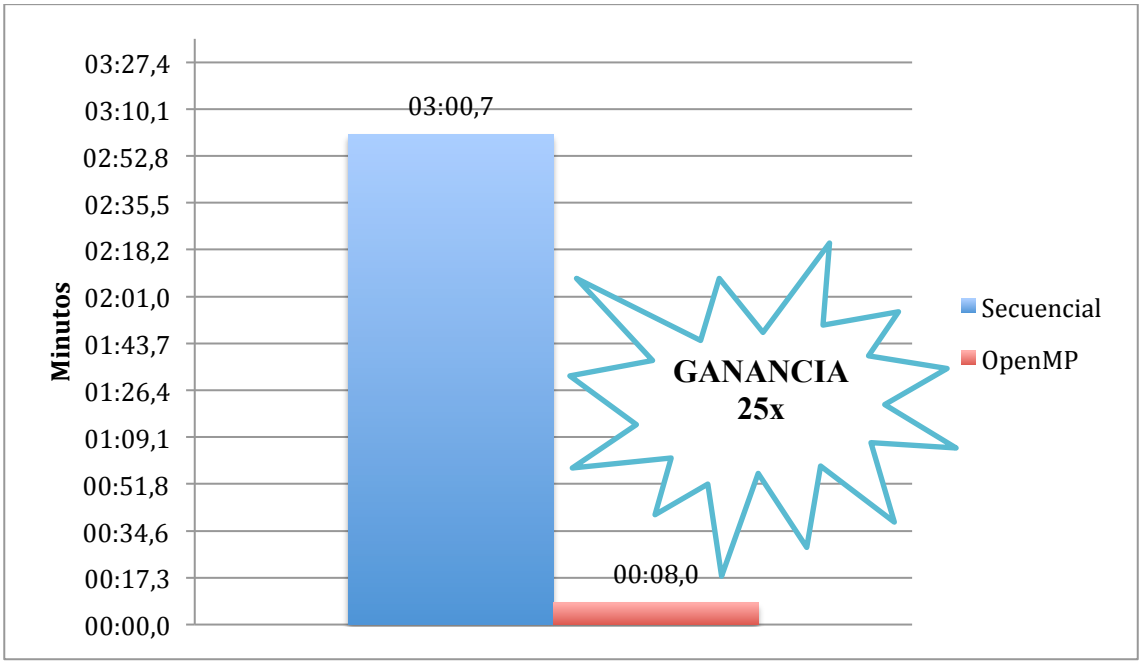
Para solucionar el problema de la falsa compartición se decidió que cada hilo mantuviera la acumulación de coincidencias de la segunda fase de forma privada durante la ejecución y, solo una vez hubieran terminado todo el trabajo, hicieran el volcado de los resultados en la memoria compartida. De esta forma, se disminuye significativamente la falsa compartición de bloques; y la competición entre hilos.

Los resultados tras el cambio pueden observarse en la siguiente gráfica, en la cual puede notarse una notable mejoría en el desempeño de los hilos.



Graf. 13 - Comparativa de ganancia aumentando el numero de hilos.

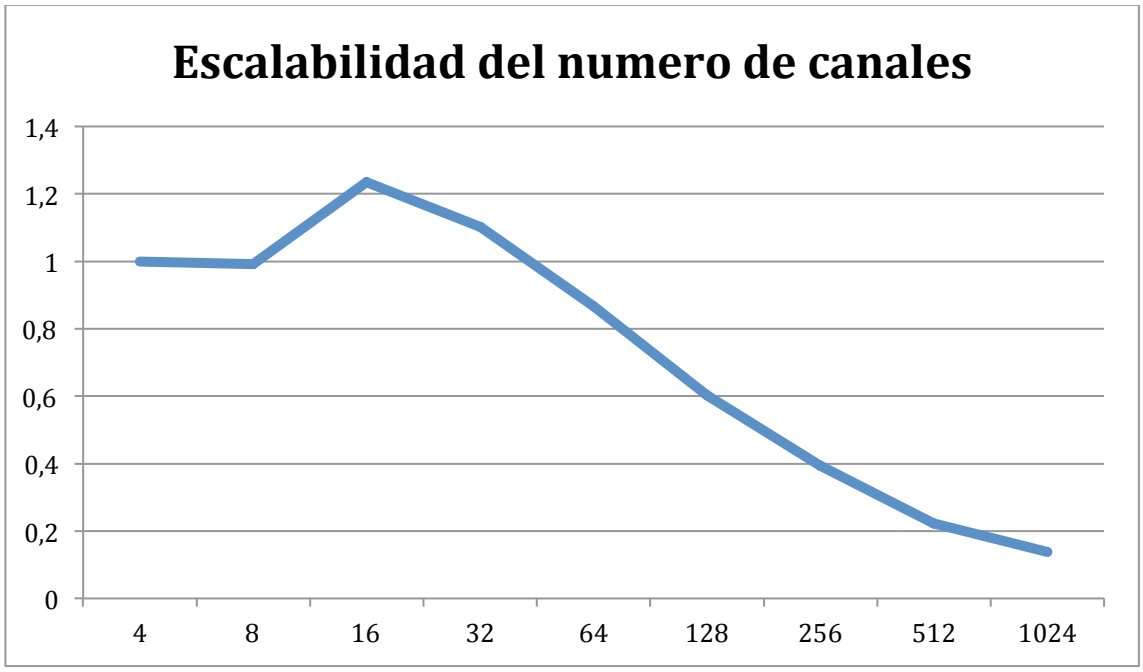
Y finalmente la comparativa entre la implementación referencial mejorada y la versión con OpenMP corregida.



Graf. 14 - Comparativa entre la versión secuencial y la versión multihilo.

4.2.4. ANÁLISIS DE ESCALABILIDAD DE DATOS

En la siguiente grafica podemos ver como se ve afectado el rendimiento del programa en base al numero de canales del problema. Para la prueba se usara una entrada de datos de 128000 muestras de tiempo y un numero de canal variable, para las medidas de rendimiento se tomara como base la ejecución con 4 canales. El experimento se ha llevado a cabo en la maquina Magnolio referenciada en el ANEXO.



Graf. 15 - Representación del rendimiento según el numero de canales.

Como puede verse en la grafica a partir de 4 canales el rendimiento va incrementándose hasta llegar a un máximo en los 32 canales tras el cual el rendimiento se reduce.

4.2.4. CONCLUSIONES

Aunque OpenMP proporciona una interfaz simple y flexible para que los programadores puedan acelerar sus programas por medio del lanzamiento de múltiples hilos para que trabajen en paralelo, hemos visto que aun así se requiere de ciertos conocimientos para lograr un buen desempeño de los hilos y que no se estorben entre ellos; además se poder localizar correctamente el mejor punto para declarar la región paralela.

He de decir, que aunque la aceleración conseguida por el programa en la maquina Magnolio es significativamente notable sigue sin conseguirse una ganancia completamente proporcional al numero de hilos involucrados. Además, cabe destacar

que, Magnolio es un ordenador con dos procesadores de 12 núcleos, es una maquina cara y difícilmente estará al alcance de los investigadores que vayan usar este programa, los cuales probablemente tengan un ordenador con 2 o 4 núcleos máximo; salvo que paguen por tiempo de servicio en algún centro de supercomputación, lo cual también es bastante caro.

4.3. PARALELIZACION CON CUDA

4.3.1. ¿QUÉ PARALELIZAR?

Para poder llevar a cabo un fructuoso traspaso del algoritmo a la arquitectura de las GPU hemos de tener claro cuales son los puntos fuertes y puntos flacos de esta arquitectura, para poder decidir que parte del algoritmo podrá beneficiarse mas de ella.

Sabiendo que las GPUs no soportan bien el uso de divergencias en la ejecución del programa, es correcto suponer que, la parte de los casos extremos del algoritmo, en el cual la ejecución esta próxima al principio y al final de la lectura de datos y a la que nos hemos referido en etapas anteriores como *exterior*, no merece la pena ser portada. Pues esta parte supone un tiempo muy reducido del tiempo de ejecución y implicaría cargar de complejidad innecesaria al programa.

Por ello decidí en un primer momento que la parte exterior se mantendría en la CPU siendo paralelizada mediante OpenMP y la parte interior seria la que se llevara al dispositivo GPU.

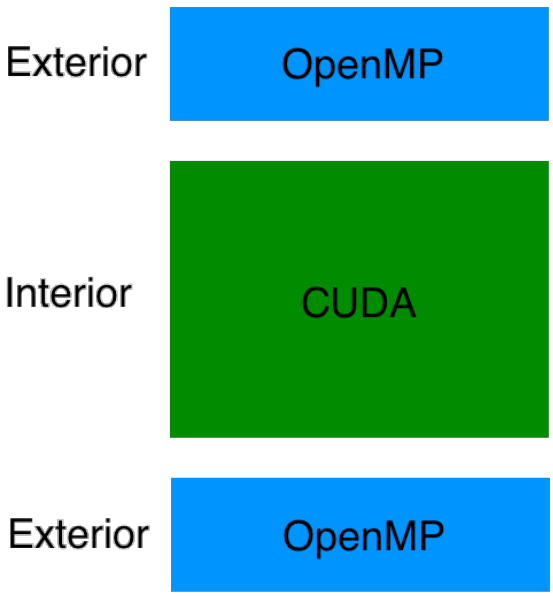


Fig. 16 - Representación de la tecnología de paralelización usada en cada parte del algoritmo.

4.3.2. DEFINICIÓN DE LOS KERNEL Y DISTRIBUCION DE HILOS

Una vez decidido que sección del algoritmo vamos a trasladar para que sea procesado en la GPU mediante CUDA, debemos decidir como vamos a distribuir los hilos y si fuera necesario, donde insertar los puntos de sincronización.

En el algoritmo SL hay dos partes diferenciadas, a los cuales me he referido como análisis de un canal y comparación de canales; debido a que la segunda requiere de los resultados de haber hecho la primera para todos los canales, decidí que era necesario un punto de sincronización entre la primera fase y la segunda. Para ello decidí separar cada fase como un Kernel separado, pues supone una barrera implícita y además me permitiría lanzar una distribución de hilos diferente para cada Kernel si fuera necesaria.

A partir de ahora y mientras hable de CUDA me referiré a la fase de análisis de un canal como Kernel 1 y a la fase de comparación de canales como Kernel 2.

A la hora de lanzar un Kernel, se le asigna la distribución de hilos que se va a usar, y las direcciones de memoria de la GPU a los que va a tener acceso, además de los parámetros del algoritmo SL necesarios para la ejecución.

Como las GPUs tienen un limite de memoria y al contrario que la CPU no pueden acceder a los recursos del disco duro, realizamos la ejecución del algoritmo a cachos por numero de instantes de tiempo; de no hacerlo así, los resultados del análisis del Kernel 1 necesarios para el Kernel 2, llenarían la memoria de la GPU a los pocos cientos o miles de instantes analizados, según la memoria de la GPU usada. El tamaño de las porciones de los datos que se van a analizar en el Kernel dependerá del numero de hilos lanzado.

En distribución de los hilos decidí que me aprovecharía de la jerarquía de hilos provista por CUDA, de forma que los hilos están identificados dentro de un bloque. Por ello, el bloque al que pertenezcan los hilos identifica también que canal esta analizando; y el identificador de hilo dentro de un bloque hace referencia al instante de tiempo, relativo al instante inicial sobre los datos que trabajaran los Kernel, que va a analizar.

Para que el programa decida el numero de hilos que va a utilizar se basara en las características del dispositivo sobre el que se va a trabajar. Para poder tener una distribución de hilos equilibrada hay que tener en cuenta el numero máximo de hilos que pueden albergar un multiprocesador o SM, el máximo de hilos por bloque y el hecho de que cada multiprocesador puede alojar un máximo de 8 bloques al mismo tiempo.

Código del calculo de la distribución de hilos

```
// K es el numero de canales del problema

// Numero de bloques
parametros->nBlocks = (K < maxBlocks ? K : maxBlocks);

// No se usa la dimensión Y de los hilos, por lo que se deja constante
parametros->tamY = 1;
```

```
// Calculamos el numero de hilos por bloque
if(K <= maxBlocks) { // Si hay menos canales que bloques disponibles

   // Nº de hilos depende del nº de canales por multiprocesador
   nHilos = (devProp.maxThreadsPerMultiProcessor / (ceil(K/ (float)
devProp.multiProcessorCount)));
   // Calculo para maximizar la ocupación de la GPU
   nHilos = (nHilos < devProp.maxThreadsPerBlock*0.5 ? nHilos :
devProp.maxThreadsPerBlock*0.5);
}
else { // Si hay mas canales que bloques que bloques disponibles
   nHilos = (devProp.maxThreadsPerMultiProcessor/8);
}
// Numero final de hilos
parametros->tamX = nHilos;
```

Se usara la misma distribución de hilos en ambos Kernel.

Codigo de lanzamiento de Kernels

```
unsigned long int sizeSharedKernel2 = nHilos*sizeof(real_t);

/* Iniciamos las llamadas a los kernel de la GPU para que ejecuten el algoritmo */
// Lanzamos tantos Kernel como pedazos hagan falta par analizar todos los datos
for (i=Ib; i<Ie; i+=nHilos) {
   int tamX = (i+nHilos<Ie ? nHilos : Ie-i);     // nº de tiempos procesados
   //Configuracion del kernel
   dim3 dimGrid(nBlocks, 1, 1); // Dividimos los canales en bloques de la GPU
   dim3 dimBlock(tamX, tamY, 1); // Asignamos un hilo por cada tiempo

   // Kernel 1: Calcula distancias y rellena la tabla de 'closers'
   Locate_Closers
   <<<dimGrid,dimBlock>>> (i, S, nData, I0, K, L, M, W, N, W1, W2,
                           parametros->data,
                           parametros->distance,
                           parametros->closers);
   // En caso de error en el Kernel se lanza un mensaje y se aborta la ejecución
   if(cudaPeekAtLastError()!=cudaSuccess) {
      fprintf(stderr "Locate_Closers (%d/%d): Channels(%d) Blocks(%d)
Threads(x=%d) Error(%s)\n", i, I, K, dimGrid.x, dimBlock.x,
cudaGetErrorString(cudaGetLastError()));
```

```
        abort();
    }

    // Kernel 2: Realiza la comparación de canales
    Cross_Channels
    <<< dimGrid, dimBlock, sizeSharedKernel2 >>> (K, W, N,
                                                  parametros->closers,
                                                  parametros->sync_like);
    // En caso de error en el Kernel se lanza un mensaje y se aborta la ejecución
    if(cudaPeekAtLastError() != cudaSuccess) {
        fprintf(stderr "Cross_Channels (%d/%d): Channels(%d) Blocks(%d)
Threads(x=%d) Shared Memory(%lu) Error(%s)\n", i, I, K, dimGrid.x, dimBlock.x,
sizeSharedKernel2, cudaGetErrorString(cudaGetLastError()));
                abort();
    }
}
```

La distribución de trabajo se realizara de la siguiente manera.

Código de la distribución de trabajo en el Kernel

```
// k es el canal sobre el que se trabaja
// Salvo que haya menos bloques que canales (K),
// cada hilo trabajara únicamente sobre un solo canal
for(int k=blockIdx.x; k<K; k+=gridDim.x) {

        int i = (inicio + threadIdx.x);  // instante de tiempo a analizar
                                         ...
```

Los bucles que vimos en la versión CPU, salvo para casos especiales, han sido remplazados por una malla implícita o *Grid*, apreciable mediante los identificadores de hilo.

4.3.3. ALOJAMIENTO DE MEMORIA

Un correcto uso de la memoria principal de la GPU es muy importante, debido a que la transferencia de datos de la memoria principal del ordenador a la memoria principal de la GPU es muy lenta por lo que este tipo de tareas suelen dejarse para el principio y

el final de la ejecución de un programa; además la mayoría de los dispositivos del mercado suelen tener unas capacidades de almacenamiento reducidas y adquirir un dispositivo con una capacidad mas elevada incrementa mucho su precio.

Es por ello que para el algoritmo SL, reservo ciertas regiones de memoria para que el programa pueda usarlos durante la ejecución de ambos Kernel, y serán reutilizados con cada nueva iteración de los Kernel, perdiendo la memoria entre una iteración y la siguiente.

Para la reserva y liberación de memoria se han creado dos métodos los cuales se ejecutaran al principio y al final del programa, estando el método principal de ejecución del algoritmo entre ambos.

Código del programa

```
int startGPU(int K, int W, int N, double *sync_like, InfoGPU *parametros) {
   // Calculo de distribución de hilos
   // Reserva de memoria
   // Inicialización de estructuras
}

                                    …
                          Ejecución del algoritmo
                                    …
void freeGPU(int K, double *sync_like, InfoGPU *parametros) {
   // Copiado resultados
   // Liberación de memoria
}
```

La única región de memoria que permanecerá inalterada durante la ejecución es la de los datos de entrada, pues desde un principio los almacenare todos en la memoria del dispositivo usando para ello el espacio que quede disponible después de haber reservado el espacio necesario para la ejecución de los Kernel. En caso de no haber espacio suficiente para todos los datos de entrada, y por que no hay mas remedio, estos se trocearan; subiéndose los trozos según se vayan terminando de analizar los datos.

En la siguiente tabla se representan las estructuras necesarias para la ejecución de los Kernel.

Nombre	Tipo	Tamaño	Kernel	Descripción
sync_like	Double	K * K	2	Se usara para almacenar los resultados del algoritmo
closers	Short	K * K * nº Hilos	1 y 2	Almacena las posiciones de los N mas próximos
distance	Double	N * K * nº Hilos	1	Almacena las distancias de los N mas próximos
data	Double	Hasta llenar o subir todos los datos	1	Contiene las señales de entrada para el algoritmo SL

Para un mejor acceso de los hilos se realiza un alineamiento al tamaño de bloque de 512 bytes mediante a las herramientas de CUDA para la reserva de memoria.

4.3.4. Optimización

A continuación voy a explicar los principales factores que se han tenido en cuenta la hora de realizar el traslado del algoritmo SL a la arquitectura GPU mediante CUDA.

4.3.4.1. Trabajo sobre paquetes

Para aprovechar el máximo paralelismo de la GPU sin sobrecargar la memoria de la GPU ni obligar a los multi-procesadores a serializar los bloques de trabajo, se ha elegido una distribución de hilos que contempla todos los posibles escenarios; de esta forma, el numero de hilos por bloque es tal que permite una máxima ocupación sin reducir el máximo numero de bloques simultáneos permitidos en un SM. Como cada hilo se encarga de analizar un instante de tiempo determinado, es necesario dividir el problema en paquetes del tamaño del numero de hilos a ejecutar. Para cada uno de estos paquetes de ejecución se lanzaran tanto el Kernel 1 como el Kernel 2, acumulando los resultados en el dispositivo.

4.3.4.2. Mantener datos en dispositivo

Como se puede intuir en el punto anterior, cada ejecución de los Kernel de la GPU acumula los resultados del algoritmo en una región de memoria, identificada por la variable *sync_like*, reservada para ello; evitando así las transferencias intermedias. El volcado de los resultados no se realizara hasta haber concluido la ejecución del algoritmo para todos los datos de entrada del problema. Al finalizar se realizara una agregación de los resultados obtenidos en la parte interior del algoritmo, llevada a cabo en el dispositivo CUDA, con la parte exterior procesada en la CPU.

4.3.4.3. TRANSPOSICIÓN DE DATOS

La transposición de datos ya se había mencionado en como una mejora del apartado de optimización secuencial, pero es en GPU donde esto se vuelve crucial.

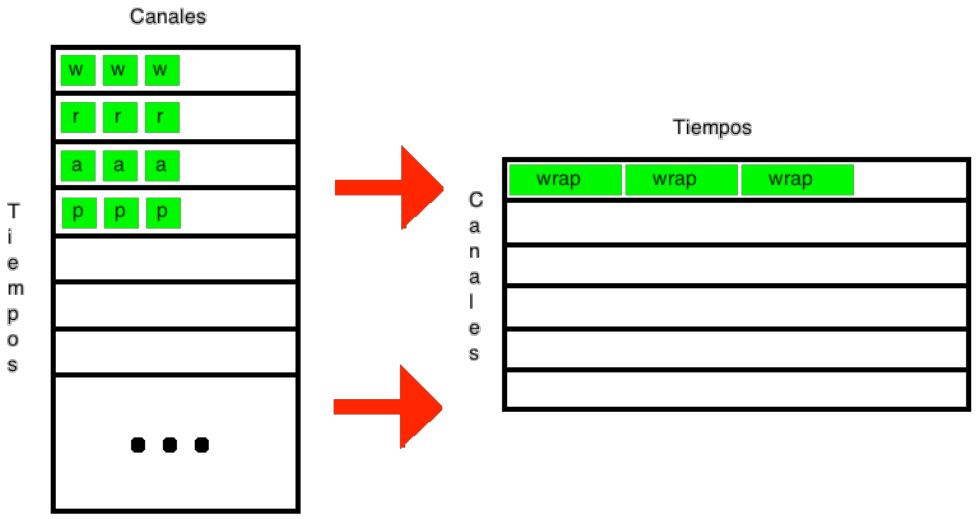


*Fig. 17 - Acceso de un **wrap** a datos.*

Los datos, tal cual se vienen en los ficheros de entrada, se leerían desde los hilos mediante accesos dispersos a memoria; provocando, en el peor de los casos, que cada hilo tratara de acceder a un bloque diferente. Al almacenar los datos transpuestos en memoria, todos hilo dentro de un mismo *wrap* accederán a direcciones contiguas en memoria, y al estar los datos alineados en bloques de 512 Bytes nos aseguramos de que cada *wrap* realiza una única transacción de bloque al realizar un acceso a memoria. De esta forma los accesos a memoria pueden aprovechar mejor el ancho de banda y competir en velocidad con los registros.

4.3.4.4. SALTOS, DIVERGENCIAS Y ACCESOS ANTICIPADOS

Al haber paralelizado con CUDA únicamente la sección interior, hemos logrado eliminar gran cantidad de saltos y divergencias de ejecución, lo cual nos permite aprovechar el paradigma de la arquitectura SIMT en la que se basan las GPUs de Nvidia; en las cuales cada Core de un SM realiza la misma instrucción.

Aun así, en se han mantenido algunas divergencias por ser inevitables. En el Kernel 1 el programa se vale de un bucle *for* que recorre vector distancia y mantiene ordenados las *N* posiciones mas próximas que componen los *closers*. Por ello se ha tratado de reducir al mínimo el cuerpo del código divergente, extrayendo de su interior aquellas operaciones que pudieran realizarse fuera.

Código divergente del Kernel 1

```
// Ordered insertion.
for (n=C; n; n--)
{
   int pos = n*blockDim.x;

   // Advance Memory Reading
   short closer = closerssp[pos-blockDim.x];
   double dist = distancesp[pos-blockDim.x];

   if (sum >= dist)
      break;
   if (n < N) {
      distancesp[pos] = dist;
      closerssp[pos] = closer;
   }
}
// Aunque se usa la estructura if-else en los dos siguientes casos no se crea divergencia
if (n < N) {
   int pos = n*blockDim.x;
   distancesp[pos] = sum;
   closerssp[pos] = w;
}
// Remember only the N closer pairs.
if (C < N) C++;
```

Además se realizan lecturas a memoria de forma anticipada al resultado de los saltos condicionales, para un máximo aprovechamiento de los ciclos de decisión.

En el Kernel 2 se realiza una comparación entre canales, para lo cual se usa una anidación de múltiples bucles los cuales se encargan de recorrer k canales, para todos los canales h, para todos los N elementos n, contra todos los N elementos x; lo que implica gran cantidad de saltos. Para tratar de reducir el numero de saltos del programa he implementación un desenrollado parcial del bucle mas interno, obligando a realizar 8 iteraciones por cada salto[5][6].

Código del bucle sin desenrollar

```
for int k= blockIdx.x; k < K; k+= gridDim.x)
{
                                                    …
   for (int h=0; h<k; h++)
   {
                                                    …
      for (int n=0;n<N;n++)
```

```
    {
                                                    …
        for (int x=0;x<N;x++)
        {
            suma+= (closeK == closerssp2[x*blockDim.x];);
         }
                                                    …
    }
                                                    …
}
```

Código del bucle con desenrollado en múltiplos de 8

```
for int k= blockIdx.x; k < K; k+= gridDim.x)
{
                                                    …
  for (int h=0; h<k; h++)
  {
                                                    …
    // Unroll loop
    int closeK = closerssp[n*blockDim.x];
    int closeH[8];
    for (int x=0;x<(N-7);x+=8)
    {
        int op = x*blockDim.x;
        closeH[0] = closerssp2[op];
        closeH[1] = closerssp2[op+=blockDim.x];
        closeH[2] = closerssp2[op+=blockDim.x];
        closeH[3] = closerssp2[op+=blockDim.x];
        closeH[4] = closerssp2[op+=blockDim.x];
        closeH[5] = closerssp2[op+=blockDim.x];
        closeH[6] = closerssp2[op+=blockDim.x];
        closeH[7] = closerssp2[op+=blockDim.x];
        suma+= (closeK == closeH[0]);
        suma+= (closeK == closeH[1]);
        suma+= (closeK == closeH[2]);
        suma+= (closeK == closeH[3]);
        suma+= (closeK == closeH[4]);
        suma+= (closeK == closeH[5]);
        suma+= (closeK == closeH[6]);
        suma+= (closeK == closeH[7]);
    }
    // Nos aseguramos las iteraciones restantes
    switch(N & 0x7) {
        case 7: closeH[6] = closerssp2[(N-7)*blockDim.x];
        case 6: closeH[5] = closerssp2[(N-6)*blockDim.x];
```

```
            case 5: closeH[4] = closerssp2[(N-5)*blockDim.x];
            case 4: closeH[3] = closerssp2[(N-4)*blockDim.x];
            case 3: closeH[2] = closerssp2[(N-3)*blockDim.x];
            case 2: closeH[1] = closerssp2[(N-2)*blockDim.x];
            case 1: closeH[0] = closerssp2[(N-1)*blockDim.x];
            case 0: ;
        }
        switch(N & 0x7) {
            case 7: suma+= (closeK == closeH[6]);
            case 6: suma+= (closeK == closeH[5]);
            case 5: suma+= (closeK == closeH[4]);
            case 4: suma+= (closeK == closeH[3]);
            case 3: suma+= (closeK == closeH[2]);
            case 2: suma+= (closeK == closeH[1]);
            case 1: suma+= (closeK == closeH[0]);
            case 0: ;
        }
                                                    …
    }
                                                    …
}
```

Aunque a simple vista el código es mas complejo y difícil de leer, permite que el programa ahorre saltos del bucle mas interno, es decir el aquel que mas veces se repite, lo cual supone un gran alivio de ciclos para el procesador. Las lecturas a memoria se realizan a nivel de *wrap* y se copian a registros; y termina con la acumulación de los resultados en el registro *suma*, lo cual es una operación muy rápida.

Para permitir el uso de un numero de canales no múltiplo de 8 se hace uso de la estructura *switch-case* para que se ocupe de los casos finales.

4.3.4.5. ACUMULACIÓN COORDINADA DE DATOS Y MEMORIA COMPARTIDA

Al finalizar el Kernel 2 se deben agregar los resultados a la solución global, en este punto surge un problema de concurrencia; pues una enorme cantidad de hilos deben escribir en una misma región de memoria.

Los hilos que se ocupan de un canal escribirán solo en la memoria referente a ese canal, por lo que no es necesario controlar los hilos cuando trabajan sobre distintos canales. Como ya he explicado los hilos están distribuidos de tal forma que todos los hilos de un canal pertenecen a un mismo bloque, por lo que deberemos ocuparnos de que todos los hilos del bloque trabajen de forma sincronizada para copiar los datos en la matriz de resultados. Para ello he hecho uso de un ejemplo de *reduce* descrito en la documentación de Nvidia para la programación en CUDA[2].

Este método hace uso de la memoria compartida de un SM y se aprovecha de una característica de la arquitectura; por la cual un bloque pertenece solo a un SM y en un SM se ejecutan concurrentemente 32 hilos, un *wrap*; por lo que solo tenemos que

vigilar el comportamiento concurrente de 32 hilos. Esto lo logramos realizando una agregación sincronizada por pasos y mediante barreras explicitas.

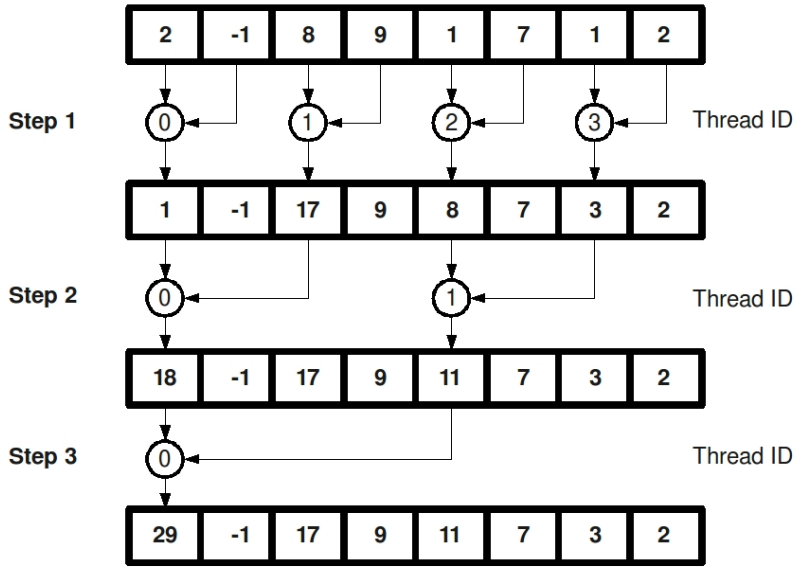


Fig. 18 - Agregación concurrente de datos de forma sincronizada.

Código de la agregación de datos

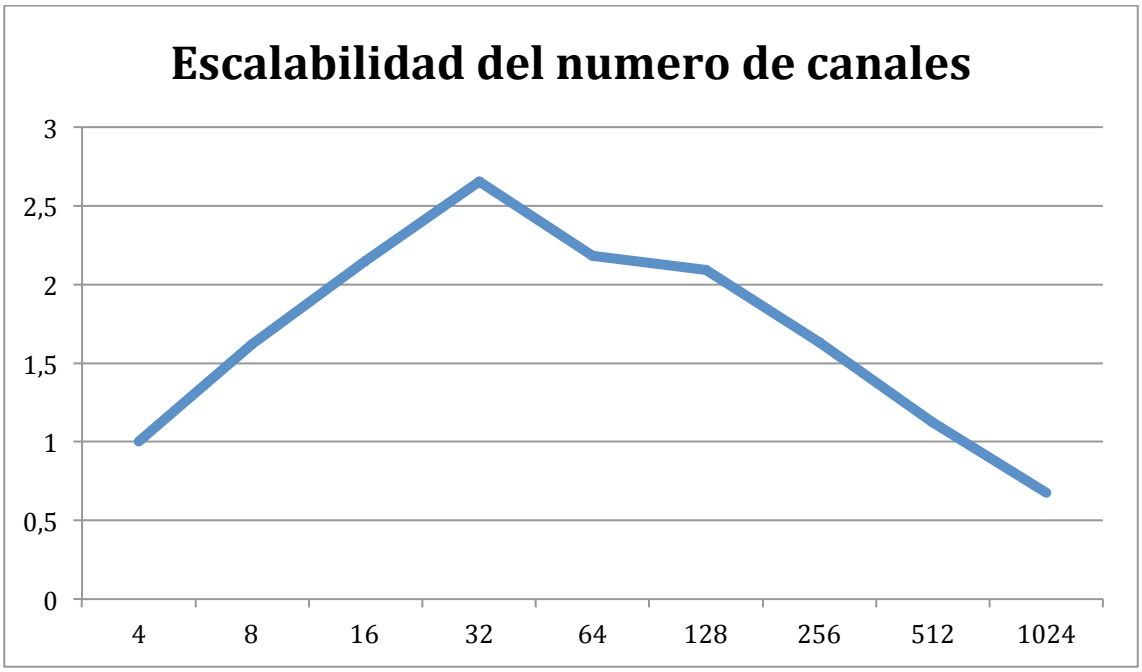
```
// Cada hilo copia el valor que quiere agregar en
// su correspondiente posición de la memoria compartida
nHitsReduce[threadIdx.x] = suma/(real_t)(N);
// barrera explicita
__syncthreads();
for(unsigned int s = 1;s<blockDim.x;s<<=1) {
   int index = s*threadIdx.x<<1;

   if(index < blockDim.x && index+s<blockDim.x)
      nHitsReduce[index]+=nHitsReduce[index + s];
   // barrera explicita
   __syncthreads();
}
// Finalmente el ultimo hilo copia el valor agregado en la matriz resultado
if(threadIdx.x == 0) {
   sync_likeKsp[h] += nHitsReduce[0];
   sync_likeHsp[k] += nHitsReduce[0];
}
```

4.3.4.6. CAPACIDAD PARA INFINIDAD DE DATOS

Para asegurarnos de que el programa soporte un problema con una cantidad infinita de datos, creamos un bucle a un nivel superior de la capa CUDA; en el cual, tras calcular la capacidad de memoria restante en la GPU para su ocupación con datos de entrada, en cada iteración copia a la GPU tantos datos como sean posibles y ejecuta la capa de CUDA, el bucle realiza las iteraciones necesarias hasta completar el análisis de todos los datos del problema.

4.3.7. ANÁLISIS DE ESCALABILIDAD DE DATOS

En la siguiente grafica podemos ver como se ve afectado el rendimiento del programa en base al numero de canales del problema. Para la prueba se usara una entrada de datos de 128000 muestras de tiempo y un numero de canal variable, para las medidas de rendimiento se tomara como base la ejecución con 4 canales. El experimento se ha llevado a cabo en la maquina Espino referenciada en el ANEXO, y con el dispositivo GTX TITAN.



Graf. 16 - Representación del rendimiento según el numero de canales

En la grafica podemos ver un comportamiento similar el detectado en la versión OpenMP, incrementándose el rendimiento a medida que vamos logrando una mejor ocupación de la GPU, pero a partir de los 64 canales empieza a producirse una perdida en el rendimiento, aunque de manera mucho mas suave que la vista en la versión CPU.

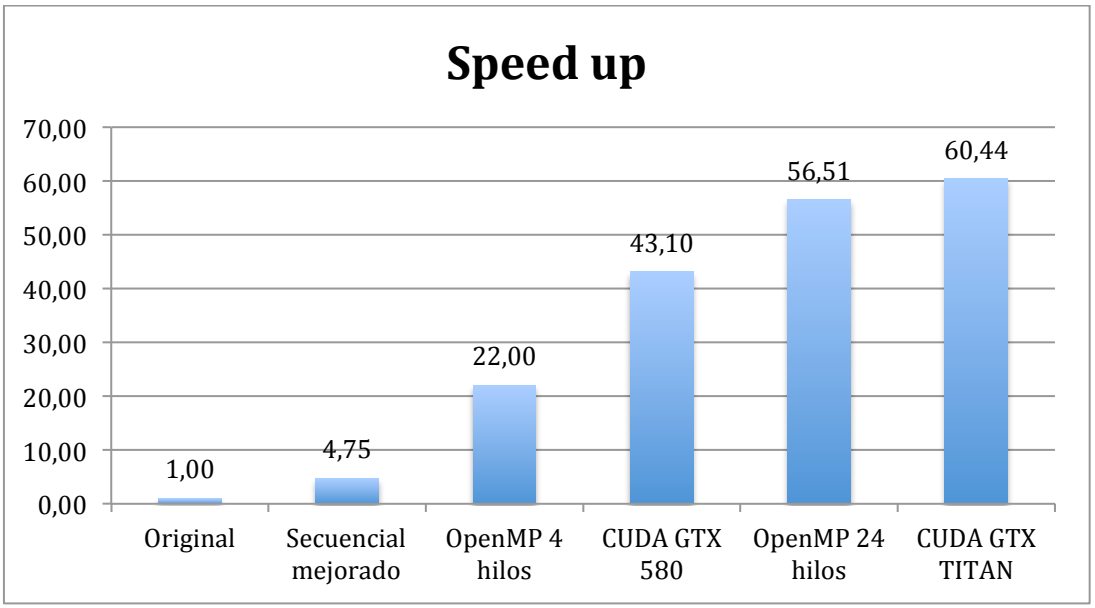
4.3.8. CONCLUSIONES

Programar para GPUs es una tarea compleja, que requiere de un conocimiento profundo del funcionamiento de este tipo de dispositivos; CUDA es una herramienta que ayuda a simplificar mucho el trabajo y lo hace de una forma amigable para el programador, para que no tenga que distanciarse demasiado de la programación clásica en CPU, a pesar de ello requiere de mucha dedicación para poder exprimir al máximo el potencial de las GPUs.

No obstante, una vez se ha logrado afinar bien un programa, CUDA permite alcanzar unas velocidades de computo muy elevadas que no podrían ser alcanzadas con CPUs convencionales; siendo necesaria la adquisición de procesadores de alta gama y elevado precio para poder competir, mientras que CUDA permite alcanzar estos umbrales usando dispositivos de gama media y un precio mucho mas asequible.

4.4. COMPARATIVA DE RENDIMIENTO

A continuación mostrare una comparativa de rendimiento con cada una de las etapas y tecnologías explicadas, haciendo uso para el experimento de la prueba *paciente_test*, que consta de 148 canales y 50865 muestras de tiempo. Todas las pruebas se han llevado a cabo en Espino, y para las pruebas de CUDA se ha usado la GPU Nvidia GeForce GTX 580.



Graf. 17 - Comparativa de distintas ejecuciones del programa.

5. CONCLUSIONES DEL PROYECTO

Entre los objetivos del proyecto nos planteamos trasladar un programa, que originalmente estaba implementado en Matlab y era demasiado lento para que sus usuarios se plantearan realmente usarlo, a un lenguaje mas rápido y de mas bajo nivel como es el lenguaje C, optimizarlo para obtener el mayor rendimiento posible, además de adaptarlo a las arquitecturas paralelas de memoria compartida y GPU; y así lo hemos hecho y evaluado tal cual se cuenta en los capítulos anteriores.

En este proyecto hemos visto como desarrollar para dos tecnologías distintas de paralelización de programas; cada una con sus ventajas e inconvenientes, y hemos realizado pruebas de rendimiento para poder elaborar una idea de las capacidades y limites de cada una de ellas.

De todo ello podemos concluir que mientras OpenMP es un método sencillo para poder paralelizar un programa, CUDA requiere un conocimiento y una complejidad que impide sea fácilmente utilizable para la gran mayoría de programadores. No obstante, las prestación alcanzadas con la tecnología CUDA podrían compensar el esfuerzo de desarrollo necesario, en tiempo y en dinero, si se va a realizar un uso intensivo del programa.

Además otra de las ventajas de la tecnología CUDA es que mientras que un procesador de altas prestaciones puede alcanzar un precio superior a los 1500€, refiriéndome a procesadores de 12 núcleos; un dispositivo GPU de Nvidia como el GeForce GTX 580 que he usado para las pruebas, tiene un valor en el mercado de apenas unos 250 € siendo posible localizarlo incluso a un menor precio.

Aun así, debido a los requerimientos de la arquitectura CUDA, antes de empezar cualquier proyecto en CUDA ha de valorarse con objetividad si el problema es realmente adaptable a las GPU y si va a lograrse un verdadero beneficio; una vez mas para poder tomar esta decisión requiere de una experiencia y conocimiento previo acerca de la tecnología.

Para concluir finalmente el documento, he de decir que para exprimir al máximo cualquier programa hace falta dedicarle mucho tiempo y esfuerzo, partiendo, no de la intuición, sino del uso de las herramientas adecuadas para cada fase.

6. ANEXO

En este apartado se describen las maquinas y dispositivos utilizadas para la realización de los diversos experimentos expuestos a lo largo del documento.

6.1. ORDENADORES

Lo siguiente es lista de las ordenadores utilizados para los diversos experimentos del proyecto y sus características principales.

6.1.1. ESPINO

Es una ordenador del grupo GOPAC, de la Facultad de Informática de la Universidad Politecnica de Madrid, que cuenta con dos procesadores Intel Xeon CPU E5645, 2.40GHz de 6 núcleos y con 48 GB de memoria RAM.

Además, este ordenador dispone de dos tarjetas Nvidia, una GTX 580 y una GTX Titan.

6.1.2. MAGNOLIO

Es una ordenador del grupo GOPAC, de la Facultad de Informática de la Universidad Politecnica de Madrid, que cuenta con cuatro procesadores AMD Opteron(tm) Processor 6176, 2,30 GHz de 12 núcleos y con 16 GB de memoria RAM.

6.2. GPUS

A continuación refiero una lista de las GPUs donde se han realizado los diversos experimentos del proyecto, así como información especifica de los dispositivos.

6.2.1. GEFORCE GTX 580

La GPU GeForce GTX 580 de la marca Nvidia tiene las siguientes características:

```
CUDA Driver Version / Runtime Version 5.5 / 5.5
CUDA Capability Major/Minor version number: 2.0
Total amount of global memory: 1536 MBytes (1610285056 bytes)
(16) Multiprocessors, ( 32) CUDA Cores/MP:  512 CUDA Cores
GPU Clock rate: 1544 MHz (1.54 GHz)
Memory Clock rate: 2004 Mhz
```

Memory Bus Width: 384-bit
L2 Cache Size: 786432 bytes
Maximum Texture Dimension Size (x,y,z) 1D=(65536), 2D=(65536, 65535), 3D=(2048, 2048, 2048)
Maximum Layered 1D Texture Size, (num) layers 1D=(16384), 2048 layers
Maximum Layered 2D Texture Size, (num) layers 2D=(16384, 16384), 2048 layers
Total amount of constant memory: 65536 bytes
Total amount of shared memory per block: 49152 bytes
Total number of registers available per block: 32768
Warp size: 32
Maximum number of threads per multiprocessor: 1536
Maximum number of threads per block: 1024
Max dimension size of a thread block (x,y,z): (1024, 1024, 64)
Max dimension size of a grid size (x,y,z): (65535, 65535, 65535)
Maximum memory pitch: 2147483647 bytes
Texture alignment: 512 bytes
Concurrent copy and kernel execution: Yes with 1 copy engine(s)
Run time limit on kernels: No
Integrated GPU sharing Host Memory: No
Support host page-locked memory mapping: Yes
Alignment requirement for Surfaces: Yes
Device has ECC support: Disabled
Device supports Unified Addressing (UVA): Yes

6.2.2. GeForce GTX Titan

La GPU GeForce GTX Titan de la marca Nvidia tiene las siguientes características:

CUDA Driver Version / Runtime Version 5.5 / 5.5
CUDA Capability Major/Minor version number: 3.5
Total amount of global memory: 6144 MBytes (6442254336 bytes)
(14) Multiprocessors, (192) CUDA Cores/MP: 2688 CUDA Cores
GPU Clock rate: 980 MHz (0.98 GHz)
Memory Clock rate: 3004 Mhz
Memory Bus Width: 384-bit
L2 Cache Size: 1572864 bytes
Maximum Texture Dimension Size (x,y,z) 1D=(65536), 2D=(65536, 65536), 3D=(4096, 4096, 4096)
Maximum Layered 1D Texture Size, (num) layers 1D=(16384), 2048 layers
Maximum Layered 2D Texture Size, (num) layers 2D=(16384, 16384), 2048 layers
Total amount of constant memory: 65536 bytes
Total amount of shared memory per block: 49152 bytes
Total number of registers available per block: 65536
Warp size: 32
Maximum number of threads per multiprocessor: 2048
Maximum number of threads per block: 1024

Max dimension size of a thread block (x,y,z): (1024, 1024, 64)
Max dimension size of a grid size (x,y,z): (2147483647, 65535, 65535)
Maximum memory pitch: 2147483647 bytes
Texture alignment: 512 bytes
Concurrent copy and kernel execution: Yes with 1 copy engine(s)
Run time limit on kernels: No
Integrated GPU sharing Host Memory: No
Support host page-locked memory mapping: Yes
Alignment requirement for Surfaces: Yes
Device has ECC support: Disabled
Device supports Unified Addressing (UVA): Yes

7. BIBLIOGRAFÍA

[1] C.J. Stam, B.W. van Dijk. “Synchronization likelihood: an unbiased measure of generalized synchronization in multivariate data sets” : Physica D 163 (2002) 236–251.

[2] David B. Kirk, Wen-mai W. Hwu. Programming Massively Parallel Processors: a hands-on approach. Applications of GPU Computing Series. Morgan Kaufmann Publishers, 2010.

[3] NVIDIA. CUDA C Programming guide. 2012.

[4] NVIDIA. CUDA C best practices guide. 2012.

[5] Vasily Volkov, UC Berkeley. Unrolling parallel loops. 2011.

[6] Vasily Volkov, UC Berkeley. Better performance at lower occupancy. 2010.

[7] Vasily Volkov, UC Berkeley. Use registers and multiple outputs per thread on GPU. 2010.

[8] Wikipedia Site.

[9] Stack OverFlow Site.

[10] The OpenMP Site.

[11] OpenACC Site.

Este documento esta firmado por

	**Firmante**	CN=tfgm.fi.upm.es, OU=CCFI, O=Facultad de Informatica - UPM, C=ES
	Fecha/Hora	Wed Jan 08 23:06:04 CET 2014
	Emisor del Certificado	EMAILADDRESS=camanager@fi.upm.es, CN=CA Facultad de Informatica, O=Facultad de Informatica - UPM, C=ES
	Numero de Serie	630
	Metodo	urn:adobe.com:Adobe.PPKLite:adbe.pkcs7.sha1 (Adobe Signature)